R Avanzado

Hadley Wickham

Table of contents

Bienvenida

Este es el sitio web de la segunda edición de **"R Avanzado"**, un libro de la Serie R de Chapman & Hall. El libro está diseñado principalmente para usuarios de R que desean mejorar sus habilidades de programación y comprensión del lenguaje. También debería ser útil para los programadores que llegan a R desde otros lenguajes, ya que les ayuda a comprender por qué R funciona de la forma en que lo hace.

Si está buscando la primera edición, puede encontrarla en http://adv-r.had.co.nz/.

Licencia

Otros libros

Usted también podría estar interesado en:

- **"Advanced R Solutions"** de Malte Grosser y Henning Bumann, proporciona soluciones elaboradas para los ejercicios de este libro.
- **"R para ciencia de datos"** que le presenta R como una herramienta para hacer ciencia de datos, centrándose en un conjunto consistente de paquetes conocido como tidyverse.
- **"R Packages"** que le enseña cómo aprovechar al máximo el fantástico sistema de paquetes de R.

Sobre la traducción

Esta traducción de “R Avanzado” es un proyecto personal de David Díaz Rodríguez con el objetivo de facilitar el estudio de conceptos avanzados de programación con R, tanto al propio traductor como a todas aquellas personas de habla hispana que deseen aprender conceptos más avanzados sobre programación con R.

Señalar que esta es una traducción textual del libro por lo que cuando el autor se refiere a sí mismo en primera persona , será Hadley Wickham y no el traductor.

La traducción fue realizada usando Google Translate y se corrigieron algunos errores gramaticales y de coherencia. Si detecta algún error relacionado con contenido de la traducción, siéntase libre de abrir un issue o un pull request en este repositorio.

Prefacio

Bienvenido a la segunda edición de *R Avanzado*. Tenía tres objetivos principales para esta edición:

- Mejorar la cobertura de conceptos importantes que entendí completamente solo después de la publicación de la primera edición.
- Reducir el tiempo de cobertura de temas que han demostrado ser menos útiles, o que creo que son realmente apasionantes pero resultan no ser tan prácticos.
- En general, hacer que el material sea más fácil de entender con un mejor texto, un código más claro y muchos más diagramas.

Si está familiarizado con la primera edición, este prefacio describe los principales cambios para que pueda enfocar su lectura en las nuevas áreas. Si está leyendo una versión impresa de este libro, notará un gran cambio muy rápidamente: ¡*R Avanzado* ahora está en color! Esto ha mejorado considerablemente el resaltado de sintaxis de los fragmentos de código y ha facilitado mucho la creación de diagramas útiles. Aproveché esto e incluí más de 100 diagramas nuevos en todo el libro.

Otro gran cambio en esta versión es el uso de nuevos paquetes, particularmente rlang, que proporciona una interfaz limpia para operaciones y estructuras de datos de bajo nivel. La primera edición usó funciones base R casi exclusivamente, lo que creó algunos desafíos pedagógicos porque muchas funciones evolucionaron de forma independiente durante varios años, lo que dificulta ver las grandes ideas subyacentes ocultas entre las variaciones incidentales en los nombres y argumentos de las funciones.

Continúo mostrando los equivalentes base en las barras laterales, notas al pie y, cuando sea necesario, en secciones individuales, pero si desea ver la expresión R base más pura de las ideas de este libro, le recomiendo leer la primera edición, que puede encontrar en línea en http://adv-r.had.co.nz.

Los fundamentos de R no han cambiado en los cinco años desde la primera edición, pero mi comprensión de ellos ciertamente sí. Por lo tanto, la estructura general de "Fundamentos" se ha mantenido más o menos igual, pero muchos de los capítulos individuales se han mejorado considerablemente:

- Capítulo 2, "Nombres y valores", es un capítulo completamente nuevo que lo ayuda a comprender la diferencia entre objetos y nombres de objetos. Esto lo ayuda a predecir con mayor precisión cuándo R hará una copia de una estructura de datos y sienta las bases importantes para comprender la programación funcional.

- Capítulo 3, "Vectores" (anteriormente denominadas estructuras de datos), se ha reescrito para centrarse en tipos de vectores como números enteros, factores y marcos de datos. Contiene más detalles de vectores S3 importantes (como fechas y fechas-horas), analiza la variación del marco de datos proporcionado por el paquete tibble (Müller and Wickham 2018) y, en general, refleja mi comprensión mejorada de los tipos de datos vectoriales.

- Capítulo 4, "Subconjunto", ahora distingue entre `[` y `[[` por su intención: `[` extrae muchos valores y `[[` extrae un solo valor (anteriormente se caracterizaban por si "simplificaban" o "conservaban"). La sección Section 4.3 dibuja el "tren" para ayudarlo a comprender cómo funciona `[[` con listas e introduce nuevas funciones que brindan un comportamiento más consistente para índices fuera de los límites.

- Capítulo 5, "Flujo de control", es un nuevo capítulo: de alguna manera me olvidé de herramientas importantes como declaraciones `if` y bucles `for`.

- Capítulo 6, "Funciones", tiene un ordenamiento mejorado, introduce las canalizaciones (`%>%` y `|>`) como una tercera forma de componer funciones (Sección Section 6.3) y ha mejorado considerablemente la cobertura de formas de funciones (Sección Section 6.8).

- Capítulo 7, "Entornos", tiene un tratamiento reorganizado de entornos especiales Section 7.4, y una discusión muy mejorada de la pila de llamadas Section 7.5.

- Capítulo 8, "Condiciones", contiene material previamente en "Excepciones y depuración", y mucho contenido nuevo sobre cómo funciona el sistema de condiciones de R. También le muestra cómo crear sus propias clases de condiciones personalizadas Section 8.5.

Los capítulos que siguen a la Parte I, Fundamentos, se han reorganizado en torno a los tres paradigmas de programación más importantes en R: programación funcional, programación orientada a objetos y metaprogramación.

- La programación funcional ahora se divide más claramente en las tres técnicas principales: "Funcionales" (Capítulo 9), "Fábricas de funciones" (Capítulo 10) y "Operadores de funciones" (Capítulo 11). Me he centrado en ideas que tienen aplicaciones prácticas en la ciencia de datos y he reducido la cantidad de teoría pura.

  Estos capítulos ahora usan funciones provistas por el paquete purrr (Henry and Wickham 2018a), lo que me permite concentrarme más en las ideas subyacentes y menos en los detalles secundarios. Esto condujo a una simplificación considerable del capítulo de operadores de funciones, ya que un uso importante era evitar la ausencia de puntos suspensivos (`...`) en los funcionales base.

- La programación orientada a objetos (POO) ahora forma una sección importante del libro con capítulos completamente nuevos sobre tipos base (Capítulo 12), S3 (Capítulo 13), S4 (Capítulo 15), R6 (Capítulo 14) y las compensaciones entre los sistemas (Capítulo 16).

Estos capítulos se enfocan en cómo funcionan los diferentes sistemas de objetos, no en cómo usarlos de manera efectiva. Esto es desafortunado, pero necesario, porque muchos de los detalles técnicos no se describen en otra parte, y el uso efectivo de OOP necesita un libro completo propio.

- La metaprogramación (anteriormente llamada "computación en el lenguaje") describe el conjunto de herramientas que puede usar para generar código con código. En comparación con la primera edición, este material se ha ampliado sustancialmente y ahora se centra en la "evaluación ordenada", un conjunto de ideas y teorías que hacen que la metaprogramación sea segura, tenga buenos principios y sea accesible para muchos más programadores de R. Capítulo 17, "Panorama general" establece de forma aproximada cómo encajan todas las piezas; Capítulo 18, "Expresiones", describe las estructuras de datos subyacentes; Capítulo 19, "Cuasicita", cubre las citas y las no comillas; Capítulo 20, "Evaluación", explica la evaluación del código en entornos especiales; y Capítulo 21, "Traducciones", reúne todos los temas para mostrar cómo puede traducir de un lenguaje (de programación) a otro.

La sección final del libro reúne los capítulos sobre técnicas de programación: creación de perfiles, medición y mejora del rendimiento y Rcpp. Los contenidos son muy similares a la primera edición, aunque la organización es un poco diferente. He realizado ligeras actualizaciones a lo largo de estos capítulos, especialmente para usar paquetes más nuevos (microbenchmark -> bench, lineprof -> profvis), pero la mayor parte del texto es el mismo.

Si bien la segunda edición ha ampliado principalmente la cobertura del material existente, se han eliminado cinco capítulos:

- El capítulo de vocabulario se eliminó porque siempre fue un poco extraño, y hay formas más efectivas de presentar listas de vocabulario que en un capítulo de libro.

- El capítulo de estilo ha sido reemplazado por una guía de estilo en línea, http://style.tidyverse.org/. La guía de estilo se combina con el nuevo paquete styler (Müller and Walthert 2018) que puede aplicar automáticamente muchas de las reglas.
- El capítulo de C se ha movido a https://github.com/hadley/r-internals, que, con el tiempo, proporcionará una guía para escribir código C que funcione con las estructuras de datos de R.
- El capítulo de memoria ha sido eliminado. Gran parte del material se ha integrado en Capítulo 2 y el resto se sintió demasiado técnico y no tan importante de entender.
- Se eliminó el capítulo sobre el desempeño de R como lenguaje. Proporcionó pocas ideas procesables y se volvió anticuado a medida que cambiaba R.

1. Introducción

Ahora he estado programando en R durante más de 15 años y lo he estado haciendo a tiempo completo durante los últimos cinco años. Esto me ha dado el lujo de tiempo para examinar cómo funciona el lenguaje. Este libro es mi intento de transmitir lo que he aprendido para que pueda comprender las complejidades de R tan rápido y sin dolor como sea posible. Leerlo te ayudará a evitar los errores que he cometido y los callejones sin salida en los que me he metido, y te enseñará herramientas, técnicas y modismos útiles que pueden ayudarte a atacar muchos tipos de problemas. En el proceso, espero mostrar que, a pesar de sus peculiaridades a veces frustrantes, R es, en el fondo, un lenguaje elegante y hermoso, bien adaptado para la ciencia de datos.

1.1. ¿Por qué R?

Si es nuevo en R, es posible que se pregunte qué hace que valga la pena aprender un lenguaje tan peculiar. Para mí, algunas de las mejores características son:

- Es gratis, de código abierto y está disponible en todas las plataformas principales. Como resultado, si realiza su análisis en R, cualquiera puede replicarlo fácilmente, independientemente de dónde viva o cuánto dinero gane.

- R tiene una comunidad diversa y acogedora, tanto en línea (por ejemplo, la comunidad de Twitter #rstats) como en persona (como las muchas reuniones de R). Dos grupos comunitarios particularmente inspiradores son rweekly newsletter que facilita mantenerse al día con R, y R-Ladies que ha creado una comunidad maravillosamente acogedora para mujeres y otros géneros minoritarios.

- Un conjunto masivo de paquetes para modelado estadístico, aprendizaje automático, visualización e importación y manipulación de datos. Sea cual sea el modelo o el gráfico que esté tratando de hacer,

lo más probable es que alguien ya haya intentado hacerlo y pueda aprender de sus esfuerzos.

- Potentes herramientas para comunicar sus resultados. Quarto facilita convertir sus resultados en archivos HTML, PDF, documentos de Word, presentaciones de PowerPoint, tableros y más. Shiny le permite crear hermosas aplicaciones interactivas sin ningún conocimiento de HTML o javascript.

- RStudio, proporciona un entorno de desarrollo integrado, adaptado a las necesidades de la ciencia de datos, el análisis interactivo de datos y la programación estadística.

- Herramientas de vanguardia. Los investigadores en estadística y aprendizaje automático suelen publicar un paquete R para acompañar sus artículos. Esto significa acceso inmediato a las últimas técnicas e implementaciones estadísticas.

- Soporte de lenguaje profundamente arraigado para el análisis de datos. Esto incluye funciones como valores perdidos, marcos de datos y vectorización.

- Una base sólida de programación funcional. Las ideas de la programación funcional se adaptan bien a los desafíos de la ciencia de datos, y el lenguaje R es funcional en el fondo y proporciona muchas primitivas necesarias para una programación funcional efectiva.

- Posit, que gana dinero vendiendo productos profesionales a equipos de usuarios de R, y da la vuelta e invierte gran parte de ese dinero en la comunidad de código abierto (más del 50 % de los ingenieros de software de Posit trabajan en proyectos de código abierto). Trabajo para Posit porque creo fundamentalmente en su misión.

- Potentes instalaciones de metaprogramación. Las capacidades de metaprogramación de R le permiten escribir funciones mágicamente sucintas y concisas y proporcionan un entorno excelente para diseñar

lenguajes específicos de dominio como ggplot2, dplyr, data.table y más.

- La facilidad con la que R puede conectarse a lenguajes de programación de alto rendimiento como C, Fortran y C++.

Por supuesto, R no es perfecto. El mayor desafío (¡y oportunidad!) de R es que la mayoría de los usuarios de R no son programadores. Esto significa que:

- Gran parte del código R que verá en la naturaleza está escrito a toda prisa para resolver un problema apremiante. Como resultado, el código no es muy elegante, rápido o fácil de entender. La mayoría de los usuarios no revisan su código para abordar estas deficiencias.

- En comparación con otros lenguajes de programación, la comunidad R está más enfocada en los resultados que en los procesos. El conocimiento de las mejores prácticas de ingeniería de software es irregular. Por ejemplo, no hay suficientes programadores de R que usen control de código fuente o pruebas automatizadas.

- La metaprogramación es un arma de doble filo. Demasiadas funciones de R usan trucos para reducir la cantidad de escritura a costa de crear un código que es difícil de entender y que puede fallar de formas inesperadas.

- La incoherencia abunda entre los paquetes contribuidos, e incluso dentro de la base R. Cada vez que usa R, se enfrenta a más de 25 años de evolución, y esto puede dificultar el aprendizaje de R porque hay muchos casos especiales que recordar.

- R no es un lenguaje de programación particularmente rápido, y el código R mal escrito puede ser terriblemente lento. R también es un usuario derrochador de la memoria.

Personalmente, creo que estos desafíos crean una gran oportunidad para que los programadores experimentados tengan un profundo impacto positivo en R y en la comunidad de R. Los usuarios de R se preocupan por escribir código de alta calidad, en particular para la investigación reproducible, pero aún no tienen las habilidades para hacerlo. Espero que este libro no solo ayude a más usuarios de R a convertirse en programadores de R, sino que también anime a los programadores de otros lenguajes a contribuir con R.

1.2. ¿Quién debería leer este libro?

Este libro está dirigido a dos públicos complementarios:

- Programadores intermedios de R que quieran profundizar en R, comprender cómo funciona el lenguaje y aprender nuevas estrategias para resolver diversos problemas.
- Programadores de otros lenguajes que están aprendiendo R y quieren entender por qué R funciona de la forma en que lo hace.

Para aprovechar al máximo este libro, deberá haber escrito una cantidad decente de código en R u otro lenguaje de programación. Debe estar familiarizado con los conceptos básicos del análisis de datos (es decir, importación, manipulación y visualización de datos), haber escrito una serie de funciones y estar familiarizado con la instalación y el uso de paquetes CRAN.

Este libro recorre la estrecha línea entre ser un libro de referencia (utilizado principalmente para búsquedas) y ser legible linealmente. Esto implica algunas compensaciones, porque es difícil linealizar el material sin dejar de mantener juntos los materiales relacionados, y algunos conceptos son mucho más fáciles de explicar si ya está familiarizado con el vocabulario técnico específico. He tratado de usar notas al pie y referencias cruzadas

para asegurarme de que aún pueda tener sentido incluso si solo sumerge los dedos de los pies en un capítulo.

1.3. ¿Qué obtendrás de este libro?

Este libro brinda el conocimiento que creo que un programador avanzado de R debe poseer: una comprensión profunda de los fundamentos junto con un amplio vocabulario que significa que puede aprender tácticamente más sobre un tema cuando sea necesario.

Después de leer este libro, usted:

- Estará familiarizado con los fundamentos de R. Comprenderá los tipos de datos complejos y las mejores formas de realizar operaciones en ellos. Tendrá una comprensión profunda de cómo funcionan las funciones, sabrá qué son los entornos y cómo hacer uso del sistema de condiciones.

- Comprenderá qué significa la programación funcional y por qué es una herramienta útil para la ciencia de datos. Podrá aprender rápidamente cómo usar las herramientas existentes y tener el conocimiento para crear sus propias herramientas funcionales cuando sea necesario.

- Conocerá la rica variedad de sistemas orientados a objetos de R. Estará más familiarizado con S3, pero sabrá de S4 y R6 y dónde buscar más información cuando sea necesario.

- Apreciará la espada de doble filo de la metaprogramación. Podrá crear funciones que utilicen una evaluación ordenada, ahorrando tipeo y creando código elegante para expresar operaciones importantes. También comprenderá los peligros y cuándo evitarlos.

- Tendrá una buena intuición para saber qué operaciones en R son lentas o usan mucha memoria. Sabrá cómo usar la creación de perfiles para identificar cuellos de botella en el rendimiento y sabrá lo suficiente de C++ para convertir funciones lentas de R en equivalentes rápidos de C++.

1.4. ¿Qué no aprenderás?

Este libro trata sobre R, el lenguaje de programación, no sobre R, la herramienta de análisis de datos. Si está buscando mejorar sus habilidades en ciencia de datos, le recomiendo que aprenda sobre tidyverse, una colección de paquetes consistentes desarrollados por mis colegas y yo. En este libro aprenderá las técnicas utilizadas para desarrollar los paquetes tidyverse; si desea aprender a usarlos, le recomiendo *R para la Ciencia de Datos*.

Si desea compartir su código R con otros, deberá crear un paquete R. Esto le permite agrupar el código junto con la documentación y las pruebas unitarias, y distribuirlo fácilmente a través de CRAN. En mi opinión, la forma más sencilla de desarrollar paquetes es con devtools, roxygen2, testthat y usethis. Puede aprender a usar estos paquetes para crear su propio paquete en *Paquetes de R*.

1.5. Meta-técnicas

Hay dos metatécnicas que son tremendamente útiles para mejorar tus habilidades como programador de R: leer el código fuente y adoptar una mentalidad científica.

Leer el código fuente es importante porque te ayudará a escribir mejor código. Un excelente lugar para comenzar a desarrollar esta habilidad es mirar el código fuente de las funciones y paquetes que usa con más frecuencia. Encontrará cosas que vale la pena emular en su propio código

y desarrollará un sentido del gusto por lo que hace un buen código R. También verás cosas que no te gustan, ya sea porque sus virtudes no son evidentes o porque ofende tu sensibilidad. No obstante, dicho código es valioso, porque ayuda a concretar sus opiniones sobre el código bueno y el malo.

Una mentalidad científica es extremadamente útil cuando se aprende R. Si no comprende cómo funciona algo, debe desarrollar una hipótesis, diseñar algunos experimentos, ejecutarlos y registrar los resultados. Este ejercicio es extremadamente útil ya que si no puede resolver algo y necesita ayuda, puede mostrar fácilmente a otros lo que intentó. Además, cuando aprendas la respuesta correcta, estarás mentalmente preparado para actualizar tu visión del mundo.

1.6. Lectura recomendada

Debido a que la comunidad de R se compone principalmente de científicos de datos, no de informáticos, hay relativamente pocos libros que profundicen en los fundamentos técnicos de R. En mi viaje personal para comprender R, he encontrado que es particularmente útil usar recursos de otros lenguajes de programación. R tiene aspectos de lenguajes de programación tanto funcionales como orientados a objetos (OO). Aprender cómo se expresan estos conceptos en R lo ayudará a aprovechar su conocimiento existente de otros lenguajes de programación y lo ayudará a identificar áreas en las que puede mejorar.

Para comprender por qué los sistemas de objetos de R funcionan de la forma en que lo hacen, descubrí que *La estructura e interpretación de los programas informáticos*[1]Abelson, Sussman, and Sussman (1996) es particularmente útil. Es un libro conciso pero profundo, y después de leerlo, sentí por primera vez que podía diseñar mi propio sistema orientado a

[1]Puedes leerlo en línea gratis en https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html

objetos. El libro fue mi primera introducción al paradigma encapsulado de la programación orientada a objetos que se encuentra en R y me ayudó a comprender las fortalezas y debilidades de este sistema. SICP también enseña la mentalidad funcional en la que crea funciones que son simples individualmente y que se vuelven poderosas cuando se componen juntas.

Para comprender las compensaciones que ha hecho R en comparación con otros lenguajes de programación, encontré *Conceptos, técnicas y modelos de programación de computadoras* (Van-Roy and Haridi 2004) extremadamente útil. Me ayudó a comprender que la semántica de copiar al modificar de R hace que sea sustancialmente más fácil razonar sobre el código y que, si bien su implementación actual no es particularmente eficiente, es un problema solucionable.

Si quieres aprender a ser un mejor programador, no hay mejor lugar al que acudir que *El programador pragmático* (Hunt and Thomas 1990). Este libro es independiente del lenguaje y brinda excelentes consejos sobre cómo ser un mejor programador.

1.7. Obteniendo ayuda

Actualmente, hay tres lugares principales para obtener ayuda cuando está atascado y no puede averiguar qué está causando el problema: Comunidad Posit, StackOverflow y la lista de correo R-help. Puede obtener ayuda fantástica en cada lugar, pero tienen sus propias culturas y expectativas. Por lo general, es una buena idea pasar un poco de tiempo investigando, aprendiendo sobre las expectativas de la comunidad, antes de publicar tu primera publicación.

Algunos buenos consejos generales:

- Algunos buenos consejos generales: asegúrese de tener la última versión de R y del paquete (o paquetes) con los que tiene problemas.

Puede ser que su problema sea el resultado de un error solucionado recientemente.

- Dedique algún tiempo a crear un ejemplo reproducible o reprex (de sus siglas en inglés **repr**oducible **ex**ample). Esto ayudará a que otros te ayuden y, a menudo, conduce a una solución sin preguntar a otros, porque en el proceso de hacer que el problema sea reproducible, a menudo descubres la causa raíz. Recomiendo aprender y usar el paquete reprex.

Si está buscando ayuda específica para resolver los ejercicios de este libro, las soluciones de Malte Grosser y Henning Bumann están disponibles en https://advanced-r-solutions.rbind.io.

1.8. Reconocimientos

Me gustaría agradecer a los muchos colaboradores de R-devel y R-help y, más recientemente, Stack Overflow y Posit Community. Hay demasiados para nombrarlos individualmente, pero me gustaría agradecer especialmente a Luke Tierney, John Chambers, JJ Allaire y Brian Ripley por brindarme generosamente su tiempo y corregir mis innumerables malentendidos.

Este libro fue escrito al aire libre, y los capítulos se anunciaron en twitter cuando se completó. Es verdaderamente un esfuerzo de la comunidad: muchas personas leen borradores, corrigen errores tipográficos, sugieren mejoras y contribuyen con contenido. Sin esos colaboradores, el libro no sería tan bueno como es y estoy profundamente agradecido por su ayuda. Un agradecimiento especial a Jeff Hammerbacher, Peter Li, Duncan Murdoch y Greg Wilson, quienes leyeron el libro de cabo a rabo y proporcionaron muchas correcciones y sugerencias.

A big thank you to all 386 contributors (in alphabetical order by username): Aaron Wolen (@aaronwolen), @absolutelyNoWarranty,

Adam Hunt (@adamphunt), @agrabovsky, Alexander Grueneberg (@agrueneberg), Anthony Damico (@ajdamico), James Manton (@ajdm), Aaron Schumacher (@ajschumacher), Alan Dipert (@alandipert), Alex Brown (@alexbbrown), @alexperrone, Alex Whitworth (@alexWhitworth), Alexandros Kokkalis (@alko989), @amarchin, Amelia McNamara (@AmeliaMN), Bryce Mecum (@amoeba), Andrew Laucius (@andrewla), Andrew Bray (@andrewpbray), Andrie de Vries (@andrie), Angela Li (@angela-li), @aranlunzer, Ari Lamstein (@arilamstein), @asnr, Andy Teucher (@ateucher), Albert Vilella (@avilella), baptiste (@baptiste), Brian G. Barkley (@BarkleyBG), Mara Averick (@batpigandme), Byron (@bcjaeger), Brandon Greenwell (@bgreenwell), Brandon Hurr (@bhive01), Jason Knight (@binarybana), Brett Klamer (@bklamer), Jesse Anderson (@blindjesse), Brian Mayer (@blmayer), Benjamin L. Moore (@blmoore), Brian Diggs (@BrianDiggs), Brian S. Yandell (@byandell), @carey1024, Chip Hogg (@chiphogg), Chris Muir (@ChrisMuir), Christopher Gandrud (@christophergandrud), Clay Ford (@clayford), Colin Fay (@ColinFay), @cortinah, Cameron Plouffe (@cplouffe), Carson Sievert (@cpsievert), Craig Citro (@craigcitro), Craig Grabowski (@craiggrabowski), Christopher Roach (@croach), Peter Meilstrup (@crowding), Crt Ahlin (@crtahlin), Carlos Scheidegger (@cscheid), Colin Gillespie (@csgillespie), Christopher Brown (@ctbrown), Davor Cubranic (@cubranic), Darren Cusanovich (@cusanovich), Christian G. Warden (@cwarden), Charlotte Wickham (@cwickham), Dean Attali (@daattali), Dan Sullivan (@dan87134), Daniel Barnett (@daniel-barnett), Daniel (@danielruc91), Kenny Darrell (@darrkj), Tracy Nance (@datapixie), Dave Childers (@davechilders), David Vukovic (@david-vukovic), David Rubinger (@davidrubinger), David Chudzicki (@dchudz), Deependra Dhakal (@DeependraD), Daisuke ICHIKAWA (@dichika), david kahle (@dkahle), David LeBauer (@dlebauer), David Schweizer (@dlschweizer), David Montaner (@dmontaner), @dmurdoch, Zhuoer Dong (@dongzhuoer), Doug Mitarotonda (@dougmitarotonda), Dragoș Moldovan-Grünfeld (@dragosmg), Jonathan Hill (@Dripdrop12), @drtjc, Julian During (@duju211), @duncanwadsworth, @eaurele, Dirk Eddelbuettel (@eddelbuettel), @EdFineOKL, Eduard Szöcs (@EDiLD),

Edwin Thoen (@EdwinTh), Ethan Heinzen (@eheinzen), @eijoac, Joel Schwartz (@eipi10), Eric Ronald Legrand (@elegrand), Elio Campitelli (@eliocamp), Ellis Valentiner (@ellisvalentiner), Emil Hvitfeldt (@EmilHvitfeldt), Emil Rehnberg (@EmilRehnberg), Daniel Lee (@erget), Eric C. Anderson (@eriqande), Enrico Spinielli (@espinielli), @etb, David Hajage (@eusebe), Fabian Scheipl (@fabian-s), @flammy0530, François Michonneau (@fmichonneau), Francois Pepin (@fpepin), Frank Farach (@frankfarach), @freezby, Frans van Dunné (@FvD), @fyears, @gagnagaman, Garrett Grolemund (@garrettgman), Gavin Simpson (@gavinsimpson), Brooke Anderson (@geanders), @gezakiss7, @gggtest, Gökçen Eraslan (@gokceneraslan), Josh Goldberg (@GoldbergData), Georg Russ (@gr650), @grasshoppermouse, Gregor Thomas (@gregorp), Garrett See (@gsee), Ari Friedman (@gsk3), Gunnlaugur Thor Briem (@gthb), Greg Wilson (@gvwilson), Hamed (@hamedbh), Jeff Hammerbacher (@hammer), Harley Day (@harleyday), @hassaad85, @helmingstay, Henning (@henningsway), Henrik Bengtsson (@HenrikBengtsson), Ching Boon (@hoscb), @hplieninger, Hörmet Yiltiz (@hyiltiz), Iain Dillingham (@iaindillingham), @IanKopacka, Ian Lyttle (@ijlyttle), Ilan Man (@ilanman), Imanuel Costigan (@imanuelcostigan), Thomas Bürli (@initdch), Os Keyes (@Ironholds), @irudnyts, i (@isomorphisms), Irene Steves (@isteves), Jan Gleixner (@jan-glx), Jannes Muenchow (@jannes-m), Jason Asher (@jasonasher), Jason Davies (@jasondavies), Chris (@jastingo), jcborras (@jcborras), Joe Cheng (@jcheng5), John Blischak (@jdblischak), @jeharmse, Lukas Burk (@jemus42), Jennifer (Jenny) Bryan (@jennybc), Justin Jent (@jentjr), Jeston (@JestonBlu), Josh Cook (@jhrcook), Jim Hester (@jimhester), @JimInNashville, @jimmyliu2017, Jim Vine (@jimvine), Jinlong Yang (@jinlong25), J.J. Allaire (@jjallaire), @JMHay, Jochen Van de Velde (@jochenvdv), Johann Hibschman (@johannh), John Baumgartner (@johnbaums), John Horton (@johnjosephhorton), @johnthomas12, Jon Calder (@jonmcalder), Jon Harmon (@jonthegeek), Julia Gustavsen (@jooolia), JorneBiccler (@JorneBiccler), Jeffrey Arnold (@jrnold), Joyce Robbins (@jtr13), Juan Manuel Truppia (@juancentro), @juangomezduaso, Kevin Markham (@justmarkham), john verzani (@jverzani), Michael

Kane (@kaneplusplus), Bart Kastermans (@kasterma), Kevin D'Auria (@kdauria), Karandeep Singh (@kdpsingh), Ken Williams (@kenahoo), Kendon Bell (@kendonB), Kent Johnson (@kent37), Kevin Ushey (@kevinushey), (@kfeng123), Karl Forner (@kforner), Kirill Sevastyanenko (@kirillseva), Brian Knaus (@knausb), Kirill Müller (@krlmlr), Kriti Sen Sharma (@ksens), Kai Tang ((@ktang), Kevin Wright (@kwstat), suo.lawrence.liu@gmail.com (@Lawrence-Liu), @ldfmrails, Kevin Kainan Li (@legendre6891), Rachel Severson (@leighseverson), Laurent Gatto (@lgatto), C. Jason Liang (@liangcj), Steve Lianoglou (@lianos), Yongfu Liao (@liao961120), Likan (@likanzhan), @lindbrook, Lingbing Feng (@Lingbing), Marcel Ramos (@LiNk-NY), Zhongpeng Lin (@linzhp), Lionel Henry (@lionel-), Lluís (@llrs), myq (@lrcg), Luke W Johnston (@lwjohnst86), Kevin Lynagh (@lynaghk), @MajoroMask, Malcolm Barrett (@malcolmbarrett), @mannyishere, @mascaretti, Matt (@mattbaggott), Matthew Grogan (@mattgrogan), @matthewhillary, Matthieu Gomez (@matthieugomez), Matt Malin (@mattmalin), Mauro Lepore (@maurolepore), Max Ghenis (@MaxGhenis), Maximilian Held (@maxheld83), Michal Bojanowski (@mbojan), Mark Rosenstein (@mbrmbr), Michael Sumner (@mdsumner), Jun Mei (@meijun), merkliopas (@merkliopas), mfrasco (@mfrasco), Michael Bach (@michaelbach), Michael Bishop (@MichaelMBishop), Michael Buckley (@michaelmikebuckley), Michael Quinn (@michaelquinn32), @miguelmorin, Michael (@mikekaminsky), Mine Cetinkaya-Rundel (@mine-cetinkaya-rundel), @mjsduncan, Mamoun Benghezal (@MoBeng), Matt Pettis (@mpettis), Martin Morgan (@mtmorgan), Guy Dawson (@Mullefa), Nacho Caballero (@nachocab), Natalya Rapstine (@natalya-patrikeeva), Nick Carchedi (@ncarchedi), Pascal Burkhard (@Nenuial), Noah Greifer (@ngreifer), Nicholas Vasile (@nickv9), Nikos Ignatiadis (@nignatiadis), Nina Munkholt Jakobsen (@nmjakobsen), Xavier Laviron (@norival), Nick Pullen (@nstjhp), Oge Nnadi (@ogennadi), Oliver Paisley (@oliverpaisley), Pariksheet Nanda (@omsai), Øystein Sørensen (@osorensen), Paul (@otepoti), Otho Mantegazza (@othomantegazza), Dewey Dunnington (@paleolimbot), Paola Corrales (@paocorrales), Parker Abercrombie (@parkerabercrombie), Patrick Hausmann (@patperu), Patrick Miller (@patr1ckm), Patrick

Werkmeister (@Patrick01), @paulponcet, @pdb61, Tom Crockett (@pelotom), @pengyu, Jeremiah (@perryjer1), Peter Hickey (@PeteHaitch), Phil Chalmers (@philchalmers), Jose Antonio Magaña Mesa (@picarus), Pierre Casadebaig (@picasa), Antonio Piccolboni (@piccolbo), Pierre Roudier (@pierreroudier), Poor Yorick (@pooryorick), Marie-Helene Burle (@prosoitos), Peter Schulam (@pschulam), John (@quantbo), Quyu Kong (@qykong), Ramiro Magno (@ramiromagno), Ramnath Vaidyanathan (@ramnathv), Kun Ren (@renkun-ken), Richard Reeve (@richardreeve), Richard Cotton (@richierocks), Robert M Flight (@rmflight), R. Mark Sharp (@rmsharp), Robert Krzyzanowski (@robertzk), @robiRagan, Romain François (@romainfrancois), Ross Holmberg (@rossholmberg), Ricardo Pietrobon (@rpietro), @rrunner, Ryan Walker (@rtwalker), @rubenfcasal, Rob Weyant (@rweyant), Rumen Zarev (@rzarev), Nan Wang (@sailingwave), Samuel Perreault (@samperochkin), @sbgraves237, Scott Kostyshak (@scottkosty), Scott Leishman (@scttl), Sean Hughes (@seaaan), Sean Anderson (@seananderson), Sean Carmody (@seancarmody), Sebastian (@sebastian-c), Matthew Sedaghatfar (@sedaghatfar), @see24, Sven E. Templer (@setempler), @sflippl, @shabbybanks, Steven Pav (@shabbychef), Shannon Rush (@shannonrush), S'busiso Mkhondwane (@sibusiso16), Sigfried Gold (@Sigfried), Simon O'Hanlon (@simonohanlon101), Simon Potter (@sjp), Leo Razoumov (@slonikaz), Richard M. Smith (@Smudgerville), Steve (@SplashDance), Scott Ritchie (@sritchie73), Tim Cole (@statist7), @ste-fan, @stephens999, Steve Walker (@stevencarlislewalker), Stefan Widgren (@stewid), Homer Strong (@strongh), Suman Khanal (@sumanstats), Dirk (@surmann), Sebastien Vigneau (@svigneau), Steven Nydick (@swnydick), Taekyun Kim (@taekyunk), Tal Galili (@talgalili), @Tazinho, Tyler Bradley (@tbradley1013), Tom B (@tbuckl), @tdenes, @thomasherbig, Thomas (@thomaskern), Thomas Lin Pedersen (@thomasp85), Thomas Zumbrunn (@thomaszumbrunn), Tim Waterhouse (@timwaterhouse), TJ Mahr (@tjmahr), Thomas Nagler (@tnagler), Anton Antonov (@tonytonov), Ben Torvaney (@Torvaney), Jeff Allen (@trestletech), Tyler Rinker (@trinker), Chitu Okoli (@Tripartio), Kirill Tsukanov (@tskir), Terence Teo (@tteo), Tim Triche, Jr. (@ttriche), @tyhenkaline, Tyler Ritchie

(@tylerritchie), Tyler Littlefield (@tyluRp), Varun Agrawal (@varun729), Vijay Barve (@vijaybarve), Victor (@vkryukov), Vaidotas Zemlys-Balevičius (@vzemlys), Winston Chang (@wch), Linda Chin (@wchi144), Welliton Souza (@Welliton309), Gregg Whitworth (@whitwort), Will Beasley (@wibeasley), William R Bauer (@WilCrofter), William Doane (@WilDoane), Sean Wilkinson (@wilkinson), Christof Winter (@winterschlaefer), Jake Thompson (@wjakethompson), Bill Carver (@wmc3), Wolfgang Huber (@wolfganghuber), Krishna Sankar (@xsankar), Yihui Xie (@yihui), yang (@yiluheihei), Yoni Ben-Meshulam (@yoni), @yuchouchen, Yuqi Liao (@yuqiliao), Hiroaki Yutani (@yutannihilation), Zachary Foster (@zachary-foster), @zachcp, @zackham, Sergio Oller (@zeehio), Edward Cho (@zerokarmaleft), Albert Zhao (@zxzb).

1.9. Convenciones

A lo largo de este libro utilizo `f()` para referirme a funciones, `g` para referirme a variables y parámetros de funciones, y `h/` a rutas.

Los bloques de código más grandes entremezclan la entrada y la salida. La salida se comenta (`#>`) de modo que si tiene una versión electrónica del libro, por ejemplo, https://adv-r.hadley.nz/, puede copiar y pegar fácilmente ejemplos en R.

Muchos ejemplos usan números aleatorios. Estos se hacen reproducibles mediante `set.seed(1014)`, que se ejecuta automáticamente al comienzo de cada capítulo.

1.10. Colofón

Este libro fue escrito en bookdown dentro de RStudio. El sitio web está alojado en netlify, y travis-ci lo actualiza automáticamente después de cada confirmación. La fuente completa está disponible en GitHub. El código del libro impreso se establece en inconsolata. Las imágenes de emoji en el libro impreso provienen de [Twitter Emoji] con licencia abierta (https://github.com/twitter/twemoji).

Esta versión del libro se creó con R version 4.4.0 (2024-04-24) y los siguientes paquetes.

package	version	source
bench	1.1.3	RSPM (R 4.4.0)
bookdown	0.39	RSPM (R 4.4.0)
bslib	0.7.0	RSPM (R 4.4.0)
dbplyr	2.5.0	RSPM (R 4.4.0)
desc	1.4.3	RSPM (R 4.4.0)
downlit	0.4.3	RSPM (R 4.4.0)
emo	0.0.0.9000	git (hadley/emo@3f03b11491ce3d6fc5601e210927eff73bf8e350)
ggbeeswarm	0.7.2	RSPM (R 4.4.0)
ggplot2	3.5.1	RSPM (R 4.4.0)
jsonlite	1.8.8	RSPM (R 4.4.0)
knitr	1.46	RSPM (R 4.4.0)
lobstr	1.1.2	RSPM (R 4.4.0)
memoise	2.0.1	RSPM (R 4.4.0)
png	0.1-8	RSPM (R 4.4.0)
profvis	0.3.8	RSPM (R 4.4.0)
Rcpp	1.0.12	RSPM (R 4.4.0)
rlang	1.1.3	RSPM (R 4.4.0)
RSQLite	2.3.6	RSPM (R 4.4.0)
scales	1.3.0	RSPM (R 4.4.0)

package	version	source
sessioninfo	1.2.2	RSPM (R 4.4.0)
sloop	1.0.1	RSPM (R 4.4.0)
testthat	3.2.1.1	RSPM (R 4.4.0)
tidyr	1.3.1	RSPM (R 4.4.0)
vctrs	0.6.5	RSPM (R 4.4.0)
xml2	1.3.6	RSPM (R 4.4.0)
zeallot	0.1.0	RSPM (R 4.4.0)

Part I.

Fundamentos

Introducción

Para comenzar su viaje en el dominio de R, los siguientes siete capítulos lo ayudarán a aprender los componentes básicos de R. Espero que ya haya visto muchas de estas piezas antes, pero probablemente no las haya estudiado en profundidad. Para ayudarlo a verificar su conocimiento actual, cada capítulo comienza con un cuestionario; Si responde correctamente todas las preguntas, ¡siéntase libre de pasar al siguiente capítulo!

1. El Chapter 2 le enseña sobre una distinción importante en la que probablemente no ha pensado profundamente: la diferencia entre un objeto y su nombre. Mejorar su modelo mental aquí lo ayudará a hacer mejores predicciones sobre cuándo R copia los datos y, por lo tanto, qué operaciones básicas son baratas y cuáles son caras.

2. El Chapter 3 se sumerge en los detalles de los vectores, ayudándole a aprender cómo encajan los diferentes tipos de vectores. También aprenderá sobre los atributos, que le permiten almacenar metadatos arbitrarios y forman la base de dos de los juegos de herramientas de programación orientada a objetos de R.

3. El Chapter 4 describe cómo usar subsetting para escribir código R claro, conciso y eficiente. Comprender los componentes fundamentales le permitirá resolver nuevos problemas al combinar los componentes básicos de formas novedosas.

4. El Chapter 5 presenta herramientas de control de flujo que le permiten ejecutar código solo bajo ciertas condiciones, o ejecutar código

repetidamente con entradas cambiantes. Estos incluyen las construcciones importantes `if` y `for`, así como herramientas relacionadas como `switch()` y `while`.

5. El Chapter 6 trata sobre las funciones, los bloques de construcción más importantes del código R. Aprenderá exactamente cómo funcionan, incluidas las reglas de scooping, que rigen cómo R busca valores de nombres. También aprenderá más sobre los detalles detrás de la evaluación diferida y cómo puede controlar lo que sucede cuando sale de una función.

6. El Chapter 7 describe una estructura de datos que es crucial para entender cómo funciona R, pero poco importante para el análisis de datos: el entorno. Los entornos son la estructura de datos que vincula los nombres a los valores y potencian herramientas importantes como los espacios de nombres de paquetes. A diferencia de la mayoría de los lenguajes de programación, los entornos en R son de "primera clase", lo que significa que puede manipularlos como cualquier otro objeto.

7. El Chapter 8 concluye los fundamentos de R con una exploración de "condiciones", el término genérico que se usa para describir errores, advertencias y mensajes. Seguramente los ha encontrado antes, por lo que en este capítulo aprenderá cómo señalarlos apropiadamente en sus propias funciones y cómo manejarlos cuando se les indique en otro lugar.

2. Nombres y valores

2.1. Introducción

En R, es importante comprender la distinción entre un objeto y su nombre. Si lo hace, le ayudará a:

- Predecir con mayor precisión el rendimiento y el uso de memoria de su código.
- Escribir código más rápido evitando copias accidentales, una fuente importante de código lento.
- Comprender mejor las herramientas de programación funcional de R.

El objetivo de este capítulo es ayudarlo a comprender la distinción entre nombres y valores, y cuándo R copiará un objeto.

Prueba

Responda las siguientes preguntas para ver si puede omitir este capítulo con seguridad. Puede encontrar las respuestas al final del capítulo en la Section 2.7.

1. Dado el siguiente data frame, ¿cómo creo una nueva columna llamada "3" que contenga la suma de `1` y `2`? Solo puede usar `$`, no `[[`. ¿Qué hace que `1`, `2` y `3` sean desafiantes como nombres de variables?

```
df <- data.frame(runif(3), runif(3))
names(df) <- c(1, 2)
```

2. En el siguiente código, ¿cuánta memoria ocupa `y`?

```
x <- runif(1e6)
y <- list(x, x, x)
```

3. ¿En qué línea se copia `a` en el siguiente ejemplo?

```
a <- c(1, 5, 3, 2)
b <- a
b[[1]] <- 10
```

Estructura

- La Section 2.2 lo introduce a la distinción entre nombres y valores, y explica cómo `<-` crea un vínculo, o referencia, entre un nombre y un valor.

- La Section 2.3 describe cuándo R hace una copia: cada vez que modificas un vector, es casi seguro que estás creando un nuevo vector modificado. Aprenderá a usar `tracemem()` para averiguar cuándo se produce realmente una copia. Luego, explorará las implicaciones que se aplican a las llamadas a funciones, listas, data frames y vectores de caracteres.

- La Section 2.4 explora las implicaciones de las dos secciones anteriores sobre cuánta memoria ocupa un objeto. Dado que su intuición puede estar profundamente equivocada y dado que `utils::object.size()` es lamentablemente inexacto, aprenderá a usar `lobstr::obj_size()`.

- La Section 2.5 describe las dos excepciones importantes para copiar al modificar: con entornos y valores con un solo nombre, los objetos se modifican en su lugar.
- La Section 2.6 concluye el capítulo con una discusión sobre el recolector de basura, que libera la memoria utilizada por objetos que ya no están referenciados por un nombre.

Requisitos previos

Usaremos el paquete lobstr para profundizar en la representación interna de los objetos R.

```
library(lobstr)
```

Fuentes

Los detalles de la gestión de memoria de R no están documentados en un solo lugar. Gran parte de la información de este capítulo se obtuvo de una lectura atenta de la documentación (en particular `?Memory` y `?gc`), la sección perfilado de memoria de *Escribiendo extensiones R* (R Core Team 2018b) y SEXPs de *R internals* (R Core Team 2018a). El resto lo descubrí leyendo el código fuente de C, realizando pequeños experimentos y haciendo preguntas sobre R-devel. Cualquier error es enteramente mío.

2.2. Binding basics

Considere este código:

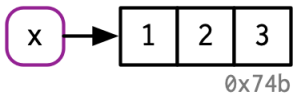
```
x <- c(1, 2, 3)
```

Es fácil leerlo como: "crear un objeto llamado 'x', que contenga los valores 1, 2 y 3". Desafortunadamente, esa es una simplificación que conducirá a predicciones inexactas sobre lo que R realmente está haciendo detrás de escena. Es más exacto decir que este código está haciendo dos cosas:

- Está creando un objeto, un vector de valores, `c(1, 2, 3)`.
- Y vincula ese objeto a un nombre, `x`.

En otras palabras, el objeto, o valor, no tiene nombre; en realidad es el nombre el que tiene un valor.

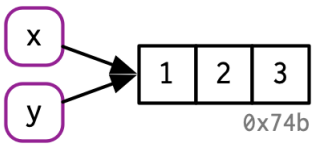
Para aclarar aún más esta distinción, dibujaré diagramas como este:



El nombre, `x`, se dibuja con un rectángulo redondeado. Tiene una flecha que apunta (o une o hace referencia) al valor, el vector `c(1, 2, 3)`. La flecha apunta en dirección opuesta a la flecha de asignación: `<-` crea un enlace desde el nombre en el lado izquierdo hasta el objeto en el lado derecho.

Por lo tanto, puede pensar en un nombre como una referencia a un valor. Por ejemplo, si ejecuta este código, no obtiene otra copia del valor `c(1, 2, 3)`, obtiene otro enlace al objeto existente:

```
y <- x
```



Es posible que hayas notado que el valor `c(1, 2, 3)` tiene una etiqueta: `0x74b`. Si bien el vector no tiene nombre, ocasionalmente necesitaré referirme a un objeto independiente de sus enlaces. Para que eso sea posible, etiquetaré los valores con un identificador único. Estos identificadores tienen una forma especial que se parece a la "dirección" de la memoria del objeto, es decir, la ubicación en la memoria donde se almacena el objeto. Pero debido a que las direcciones de memoria reales cambian cada vez que se ejecuta el código, usamos estos identificadores en su lugar.

Puede acceder al identificador de un objeto con `lobstr::obj_addr()`. Hacerlo te permite ver que tanto `x` como `y` apuntan al mismo identificador:

```
obj_addr(x)
#> [1] "0x563a18707958"
obj_addr(y)
#> [1] "0x563a18707958"
```

Estos identificadores son largos y cambian cada vez que reinicia R.

Puede tomar algún tiempo comprender la distinción entre nombres y valores, pero comprender esto es realmente útil en la programación funcional, donde las funciones pueden tener diferentes nombres en diferentes contextos.

2.2.1. Nombres no sintácticos

R tiene reglas estrictas sobre lo que constituye un nombre válido. Un nombre **sintáctico** debe constar de letras[1], dígitos, `.` y `_` pero no puede

[1] Sorprendentemente, precisamente lo que constituye una letra está determinado por su ubicación actual. Eso significa que la sintaxis del código R en realidad puede diferir de una computadora a otra, y que es posible que un archivo que funciona en una computadora ni siquiera se analice en otra. Evite este problema apegado a los caracteres ASCII (es decir, A-Z) tanto como sea posible.

comenzar con `_` o un dígito. Además, no puede usar ninguna de las **palabras reservadas** como `TRUE`, `NULL`, `if` y `function` (vea la lista completa en `?Reserved`). Un nombre que no sigue estas reglas es un nombre **no sintáctico**; si intenta usarlos, obtendrá un error:

```
_abc <- 1
#> Error: unexpected input in "_"

if <- 10
#> Error: unexpected assignment in "if <-"
```

Es posible anular estas reglas y usar cualquier nombre, es decir, cualquier secuencia de caracteres, rodeándolo con acentos graves:

```
`_abc` <- 1
`_abc`
#> [1] 1

`if` <- 10
`if`
#> [1] 10
```

Si bien es poco probable que cree deliberadamente nombres tan locos, debe comprender cómo funcionan estos nombres locos porque los encontrará, más comúnmente cuando carga datos que se han creado fuera de R.

Puede también crear enlaces no sintácticos usando comillas simples o dobles (por ejemplo, `"_abc" <- 1`) en lugar de acentos graves, pero no debería, porque tendrá que usar una sintaxis diferente para recuperar los valores. La capacidad de usar cadenas en el lado izquierdo de la flecha de asignación es un artefacto histórico, usado antes de que R admitiera los acentos graves.

2.2.2. Ejercicios

1. Explique la relación entre `a`, `b`, `c` y `d` en el siguiente código:

```
a <- 1:10
b <- a
c <- b
d <- 1:10
```

2. El siguiente código accede a la función de media de varias maneras. ¿Todos apuntan al mismo objeto de función subyacente? Verifique esto con `lobstr::obj_addr()`.

```
mean
base::mean
get("mean")
evalq(mean)
match.fun("mean")
```

3. De forma predeterminada, las funciones de importación de datos base R, como `read.csv()`, convertirán automáticamente los nombres no sintácticos en sintácticos. ¿Por qué podría ser esto problemático? ¿Qué opción le permite suprimir este comportamiento?

4. ¿Qué reglas usa `make.names()` para convertir nombres no sintácticos en sintácticos?

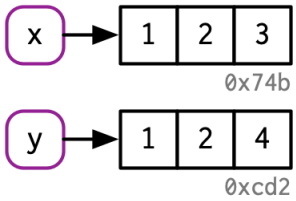
5. Simplifiqué ligeramente las reglas que rigen los nombres sintácticos. ¿Por qué `.123e1` no es un nombre sintáctico? Lea `?make.names` para obtener todos los detalles.

2.3. Copiar al modificar

Considere el siguiente código. Vincula `x` e `y` al mismo valor subyacente, luego modifica `y`[2].

```
x <- c(1, 2, 3)
y <- x

y[[3]] <- 4
x
#> [1] 1 2 3
```

Modificar `y` claramente no modificó `x`. Entonces, ¿qué pasó con el enlace compartido? Mientras que el valor asociado con `y` cambió, el objeto original no lo hizo. En su lugar, R creó un nuevo objeto, '0xcd2', una copia de '0x74b' con un valor cambiado, y luego rebotó 'y' a ese objeto.



Este comportamiento se llama **copiar al modificar**. Comprenderlo mejorará radicalmente su intuición sobre el rendimiento del código R. Una forma relacionada de describir este comportamiento es decir que los objetos R no se pueden modificar o **inmutables**. Sin embargo, generalmente

[2]Es posible que se sorprenda al ver que `[[` se usa para crear un subconjunto de un vector numérico. Volveremos a esto en la Section 4.3, pero en resumen, creo que siempre debes usar `[[` cuando obtienes o configuras un solo elemento.

evitaré ese término porque hay un par de excepciones importantes para copiar al modificar que aprenderá en la Section 2.5.

Al explorar el comportamiento de copiar al modificar de forma interactiva, tenga en cuenta que obtendrá diferentes resultados dentro de RStudio. Esto se debe a que el panel de entorno debe hacer una referencia a cada objeto para mostrar información sobre él. Esto distorsiona su exploración interactiva pero no afecta el código dentro de las funciones y, por lo tanto, no afecta el rendimiento durante el análisis de datos. Para experimentar, recomiendo ejecutar R directamente desde la terminal o usar Quarto (como este libro).

2.3.1. `tracemem()`

Puedes ver cuándo se copia un objeto con la ayuda de `base::tracemem()`. Una vez que llame a esa función con un objeto, obtendrá la dirección actual del objeto:

```
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x7f80c0e0ffc8>
```

A partir de ese momento, cada vez que se copie ese objeto, `tracemem()` imprimirá un mensaje que le indicará qué objeto se copió, su nueva dirección y la secuencia de llamadas que llevaron a la copia:

```
y <- x
y[[3]] <- 4L
#> tracemem[0x7f80c0e0ffc8 -> 0x7f80c4427f40]:
```

Si modifica `y` de nuevo, no se copiará. Esto se debe a que el nuevo objeto ahora solo tiene un único nombre vinculado, por lo que R aplica la optimización de modificación en el lugar. Volveremos a esto en la Section 2.5.

```
y[[3]] <- 5L

untracemem(x)
```

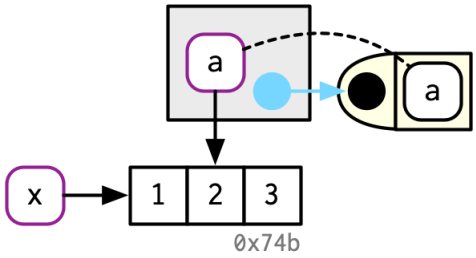
`untracemem()` es lo contrario de `tracemem()`; apaga el rastreo.

2.3.2. Llamadas de función

Las mismas reglas para copiar también se aplican a las llamadas a funciones. Toma este código:

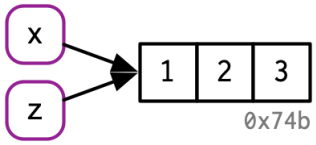
```
f <- function(a) {
  a
}

x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x563a1867be18>

z <- f(x)
# there's no copy here!

untracemem(x)
```

Mientras `f()` se está ejecutando, `a` dentro de la función apunta al mismo valor que `x` fuera de la función:



Aprenderá más sobre las convenciones utilizadas en este diagrama en la Section 7.4.4. En resumen: la función `f()` está representada por el objeto amarillo a la derecha. Tiene un argumento formal, `a`, que se convierte en un enlace (indicado por una línea negra punteada) en el entorno de ejecución (el cuadro gris) cuando se ejecuta la función.

Una vez que `f()` se complete, `x` y `z` apuntarán al mismo objeto. `0x74b` nunca se copia porque nunca se modifica. Si `f()` modificara `x`, R crearía una nueva copia y luego `z` vincularía ese objeto.

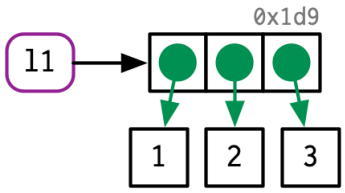


2.3.3. Listas

No son solo los nombres (es decir, las variables) los que apuntan a los valores; los elementos de las listas también lo hacen. Considere esta lista, que es superficialmente muy similar al vector numérico anterior:

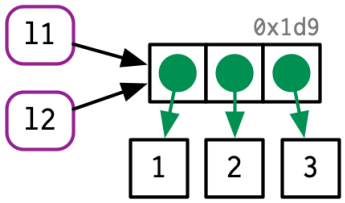
```
l1 <- list(1, 2, 3)
```

Esta lista es más compleja porque en lugar de almacenar los valores en sí, almacena referencias a ellos:

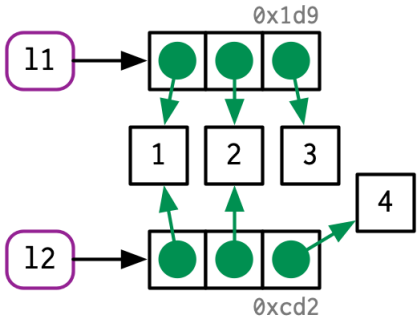


Esto es particularmente importante cuando modificamos una lista:

```
l2 <- l1
```




```
l2[[3]] <- 4
```



Al igual que los vectores, las listas utilizan el comportamiento de copiar al modificar; la lista original no se modifica y R crea una copia modificada. Esto, sin embargo, es una copia **superficial**: el objeto de la lista y sus enlaces se copian, pero los valores a los que apuntan los enlaces no. Lo opuesto a una copia superficial es una copia profunda donde se copian los contenidos de cada referencia. Antes de R 3.1.0, las copias siempre eran copias profundas.

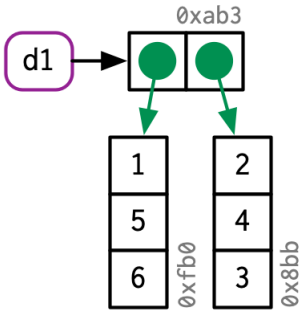
Para ver los valores que se comparten en las listas, use `lobstr::ref()`. `ref()` imprime la dirección de memoria de cada objeto, junto con una identificación local para que pueda cruzar fácilmente los componentes compartidos.

```
ref(l1, l2)
#>   [1:0x563a1740dbb8] <list>
#>  [2:0x563a14d580e8] <dbl>
#>  [3:0x563a14d58120] <dbl>
#>  [4:0x563a14d58158] <dbl>
#>
#>   [5:0x563a14cc0a68] <list>
#>  [2:0x563a14d580e8]
#>  [3:0x563a14d58120]
#>  [6:0x563a18700930] <dbl>
```

2.3.4. Data frames

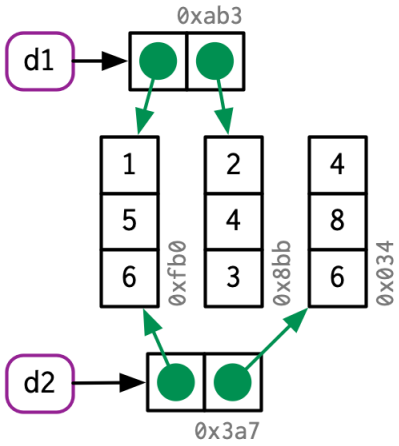
Los data frames son listas de vectores, por lo que copiar al modificar tiene consecuencias importantes cuando modifica un data frame. Tome este data frame como un ejemplo:

```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```



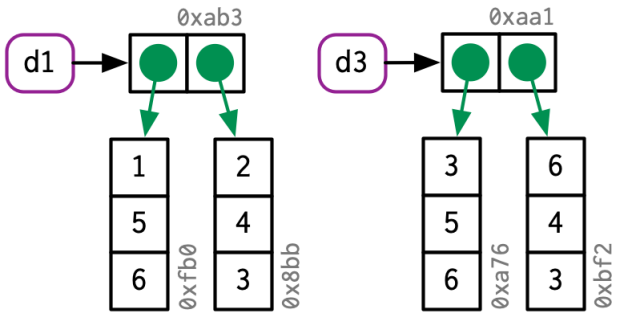
Si modifica una columna, solo *esa* columna debe modificarse; los otros seguirán apuntando a sus referencias originales:

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
```



Sin embargo, si modifica una fila, se modifican todas las columnas, lo que significa que se deben copiar todas las columnas:

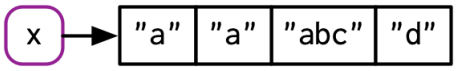
```
d3 <- d1
d3[1, ] <- d3[1, ] * 3
```



2.3.5. Vectores de caracteres

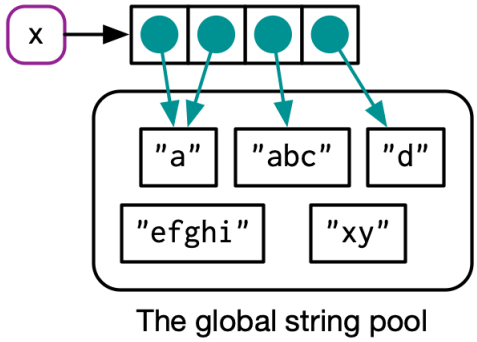
El último lugar donde R usa referencias es con vectores de caracteres [3]. Normalmente dibujo vectores de caracteres como este:

```
x <- c("a", "a", "abc", "d")
```



Pero esto es una ficción educada. R en realidad usa un **grupo de cadenas global** donde cada elemento de un vector de caracteres es un puntero a una cadena única en el grupo:

[3] Confusamente, un vector de caracteres es un vector de cadenas, no de caracteres individuales.



Puede solicitar que `ref()` muestre estas referencias configurando el argumento `character` en `TRUE`:

```
ref(x, character = TRUE)
#>   [1:0x563a173a2e48] <chr>
#>  [2:0x563a1232ec30] <string: "a">
#>  [2:0x563a1232ec30]
#>  [3:0x563a1665ff30] <string: "abc">
#>  [4:0x563a12784ae8] <string: "d">
```

Esto tiene un impacto profundo en la cantidad de memoria que usa un vector de caracteres, pero por lo demás generalmente no es importante, por lo que en otras partes del libro dibujaré vectores de caracteres como si las cadenas vivieran dentro de un vector.

2.3.6. Ejercicios

1. ¿Por qué `tracemem(1:10)` no es útil?

2. Explique por qué `tracemem()` muestra dos copias cuando ejecuta este código. Sugerencia: mire cuidadosamente la diferencia entre este código y el código que se muestra en la sección anterior.

```
x <- c(1L, 2L, 3L)
tracemem(x)

x[[3]] <- 4
```

3. Esboza la relación entre los siguientes objetos:

```
a <- 1:10
b <- list(a, a)
c <- list(b, a, 1:10)
```

4. ¿Qué sucede cuando ejecutas este código?

```
x <- list(1:10)
x[[2]] <- x
```

   Dibuja una imagen.

2.4. Tamaño del objeto

Puedes averiguar cuánta memoria ocupa un objeto con `lobstr::obj_size()`[4]:

```
obj_size(letters)
#> 1.71 kB
obj_size(ggplot2::diamonds)
#> 3.46 MB
```

Dado que los elementos de las listas son referencias a valores, el tamaño de una lista puede ser mucho más pequeño de lo esperado:

[4]Tenga cuidado con la función `utils::object.size()`. No tiene en cuenta correctamente las referencias compartidas y devolverá tamaños que son demasiado grandes.

```
x <- runif(1e6)
obj_size(x)
#> 8.00 MB

y <- list(x, x, x)
obj_size(y)
#> 8.00 MB
```

`y` es sólo 80 bytes[5] mayor que `x`. Ese es el tamaño de una lista vacía con tres elementos:

```
obj_size(list(NULL, NULL, NULL))
#> 80 B
```

Del mismo modo, debido a que R usa un grupo de cadenas global, los vectores de caracteres ocupan menos memoria de lo que cabría esperar: repetir una cadena 100 veces no hace que ocupe 100 veces más memoria.

```
banana <- "bananas bananas bananas"
obj_size(banana)
#> 136 B
obj_size(rep(banana, 100))
#> 928 B
```

Las referencias también dificultan pensar en el tamaño de los objetos individuales. `obj_size(x) + obj_size(y)` solo será igual a `obj_size(x, y)` si no hay valores compartidos. Aquí, el tamaño combinado de `x` e `y` es el mismo que el tamaño de `y`:

[5] Si está ejecutando R de 32 bits, verá tamaños ligeramente diferentes.

```
obj_size(x, y)
#> 8.00 MB
```

Finalmente, R 3.5.0 y las versiones posteriores tienen una función que podría generar sorpresas: ALTREP, abreviatura de **representación alternativa**. Esto permite que R represente ciertos tipos de vectores de forma muy compacta. El lugar donde es más probable que vea esto es con `:` porque en lugar de almacenar cada número en la secuencia, R solo almacena el primer y el último número. Esto significa que cada secuencia, sin importar cuán grande sea, tiene el mismo tamaño:

```
obj_size(1:3)
#> 680 B
obj_size(1:1e3)
#> 680 B
obj_size(1:1e6)
#> 680 B
obj_size(1:1e9)
#> 680 B
```

2.4.1. Ejercicios

1. En el siguiente ejemplo, ¿por qué `object.size(y)` y `obj_size(y)` son tan radicalmente diferentes? Consulta la documentación de `object.size()`.

```
y <- rep(list(runif(1e4)), 100)

object.size(y)
#> 8005648 bytes
obj_size(y)
#> 80.90 kB
```

2. Toma la siguiente lista. ¿Por qué su tamaño es algo engañoso?

```
funs <- list(mean, sd, var)
obj_size(funs)
#> 18.76 kB
```

3. Prediga la salida del siguiente código:

```
a <- runif(1e6)
obj_size(a)

b <- list(a, a)
obj_size(b)
obj_size(a, b)

b[[1]][[1]] <- 10
obj_size(b)
obj_size(a, b)

b[[2]][[1]] <- 10
obj_size(b)
obj_size(a, b)
```

2.5. Modificar en el lugar

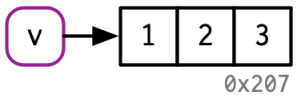
Como hemos visto anteriormente, modificar un objeto R generalmente crea una copia. Hay dos excepciones:

- Los objetos con un solo enlace obtienen una optimización de rendimiento especial.
- Los entornos, un tipo especial de objeto, siempre se modifican en su lugar.

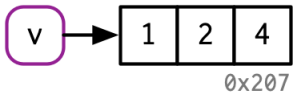
2.5.1. Objetos con un solo enlace

Si un objeto tiene un solo nombre vinculado, R lo modificará en su lugar:

```
v <- c(1, 2, 3)
```




```
v[[3]] <- 4
```



(Tenga en cuenta los ID de objeto aquí: `v` continúa enlazando con el mismo objeto, `0x207`).

Dos complicaciones hacen que predecir exactamente cuándo R aplica esta optimización sea un desafío:

- Cuando se trata de enlaces, R actualmente puede [6] solo contar 0, 1 o muchos. Eso significa que si un objeto tiene dos enlaces y uno desaparece, el recuento de referencias no vuelve a 1: uno menos que muchos sigue siendo muchos. A su vez, esto significa que R hará copias cuando a veces no sea necesario.

[6] Para cuando lea esto, es posible que esto haya cambiado, ya que hay planes en marcha para mejorar el conteo de referencias: https://developer.r-project.org/Refcnt.html

- Cada vez que llama a la gran mayoría de las funciones, hace una referencia al objeto. La única excepción son las funciones C "primitivas" especialmente escritas. Estos solo pueden ser escritos por R-core y ocurren principalmente en el paquete base.

Juntas, estas dos complicaciones hacen que sea difícil predecir si se producirá o no una copia. En cambio, es mejor determinarlo empíricamente con `tracemem()`.

Exploremos las sutilezas con un caso de estudio usando bucles for. Los bucles for tienen la reputación de ser lentos en R, pero a menudo esa lentitud se debe a que cada iteración del bucle crea una copia. Considere el siguiente código. Resta la mediana de cada columna de un data frame grande:

```
x <- data.frame(matrix(runif(5 * 1e4), ncol = 5))
medians <- vapply(x, median, numeric(1))

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

Este ciclo es sorprendentemente lento porque cada iteración del ciclo copia el data frame. Puedes ver esto usando `tracemem()`:

```
cat(tracemem(x), "\n")
#> <0x7f80c429e020>

for (i in 1:5) {
  x[[i]] <- x[[i]] - medians[[i]]
}
#> tracemem[0x7f80c429e020 -> 0x7f80c0c144d8]:
#> tracemem[0x7f80c0c144d8 -> 0x7f80c0c14540]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14540 -> 0x7f80c0c145a8]: [[<-.data.frame [[<-
```

```
#> tracemem[0x7f80c0c145a8 -> 0x7f80c0c14610]:
#> tracemem[0x7f80c0c14610 -> 0x7f80c0c14678]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14678 -> 0x7f80c0c146e0]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c146e0 -> 0x7f80c0c14748]:
#> tracemem[0x7f80c0c14748 -> 0x7f80c0c147b0]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c147b0 -> 0x7f80c0c14818]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14818 -> 0x7f80c0c14880]:
#> tracemem[0x7f80c0c14880 -> 0x7f80c0c148e8]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c148e8 -> 0x7f80c0c14950]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14950 -> 0x7f80c0c149b8]:
#> tracemem[0x7f80c0c149b8 -> 0x7f80c0c14a20]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14a20 -> 0x7f80c0c14a88]: [[<-.data.frame [[<-

untracemem(x)
```

De hecho, cada iteración copia el data frame no una, ni dos, ¡sino tres veces! Se hacen dos copias con `[[.data.frame`, y se hace otra copia[7] porque `[[.data.frame` es una función normal que incrementa el recuento de referencias de `x`.

Podemos reducir el número de copias usando una lista en lugar de un data frame. La modificación de una lista utiliza código C interno, por lo que las referencias no se incrementan y no se realiza ninguna copia:

```
y <- as.list(x)
cat(tracemem(y), "\n")
#> <0x7f80c5c3de20>

for (i in 1:5) {
  y[[i]] <- y[[i]] - medians[[i]]
}
```

[7] Estas copias son superficiales: solo copian la referencia a cada columna individual, no el contenido de las columnas. Esto significa que el rendimiento no es terrible, pero obviamente no es tan bueno como podría ser.

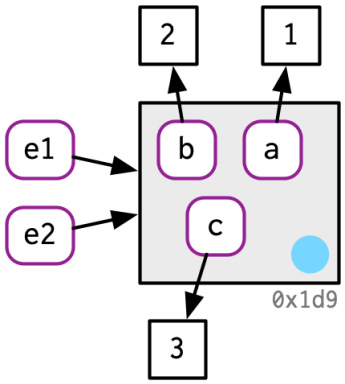
Si bien no es difícil determinar cuándo se realiza una copia, es difícil evitarlo. Si se encuentra recurriendo a trucos exóticos para evitar copias, puede ser hora de reescribir su función en C++, como se describe en el Chapter 25.

2.5.2. Entornos

Aprenderá más sobre los entornos en el Chapter 7, pero es importante mencionarlos aquí porque su comportamiento es diferente al de otros objetos: los entornos siempre se modifican en su lugar. Esta propiedad a veces se describe como **semántica de referencia** porque cuando modifica un entorno, todos los enlaces existentes a ese entorno continúan teniendo la misma referencia.

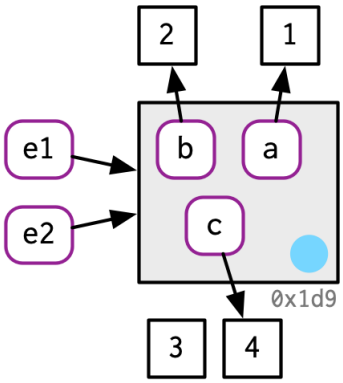
Tome este entorno, que vinculamos a `e1` y `e2`:

```
e1 <- rlang::env(a = 1, b = 2, c = 3)
e2 <- e1
```



Si cambiamos un enlace, el entorno se modifica en su lugar:

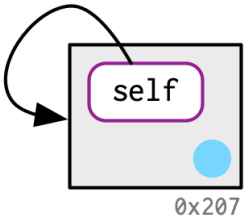
```
e1$c <- 4
e2$c
#> [1] 4
```



Esta idea básica se puede utilizar para crear funciones que "recuerden" su estado anterior. Consulte la Section 10.2.4 para obtener más detalles. Esta propiedad también se usa para implementar el sistema de programación orientado a objetos R6, el tema del Chapter 14.

Una consecuencia de esto es que los entornos pueden contenerse a sí mismos:

```
e <- rlang::env()
e$self <- e

ref(e)
#>   [1:0x563a12e626b0] <env>
#>  self = [1:0x563a12e626b0]
```



¡Esta es una propiedad única de los entornos!

2.5.3. Ejercicios

1. Explique por qué el siguiente código no crea una lista circular.

```
x <- list()
x[[1]] <- x
```

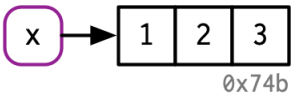
2. Envuelva los dos métodos para restar medianas en dos funciones, luego use el paquete `bench`(Hester 2018) para comparar cuidadosamente sus velocidades. ¿Cómo cambia el rendimiento a medida que aumenta el número de columnas?

3. ¿Qué sucede si intenta usar `tracemem()` en un entorno?

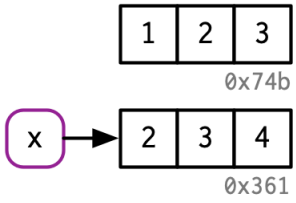
2.6. Desvincular y el recolector de basura.

Considere este código:

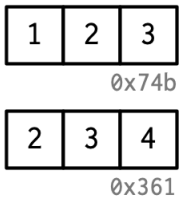
```
x <- 1:3
```



```
x <- 2:4
```




```
rm(x)
```



Creamos dos objetos, pero cuando finaliza el código, ninguno de los objetos está vinculado a un nombre. ¿Cómo se eliminan estos objetos? Ese es el trabajo del **recolector de basura**, o GC para abreviar. El GC libera memoria eliminando objetos R que ya no se usan y solicitando más memoria del sistema operativo si es necesario.

R utiliza un GC de **trazado**. Esto significa que rastrea todos los objetos a los que se puede acceder desde el entorno global[8] y todos los objetos a los que, a su vez, se puede acceder desde esos objetos (es decir, las referencias en listas y entornos se buscan de forma recursiva). El recolector de elementos no utilizados no utiliza el recuento de referencias de modificación en el lugar descrito anteriormente. Si bien estas dos ideas están

[8] Y todos los entornos de la pila de llamadas actual.

estrechamente relacionadas, las estructuras de datos internas están optimizadas para diferentes casos de uso.

El recolector de basura (GC) se ejecuta automáticamente cada vez que R necesita más memoria para crear un nuevo objeto. Mirando desde el exterior, es básicamente imposible predecir cuándo se ejecutará el GC. De hecho, ni siquiera deberías intentarlo. Si desea saber cuándo se ejecuta GC, llame a `gcinfo(TRUE)` y GC imprimirá un mensaje en la consola cada vez que se ejecute.

Puedes forzar la recolección de basura llamando a `gc()`. Pero a pesar de lo que hayas leído en otros lugares, nunca hay *necesidad* de llamar a `gc()` tú mismo. Las únicas razones por las que podría *querer* llamar a `gc()` es para pedirle a R que devuelva la memoria a su sistema operativo para que otros programas puedan usarla, o por el efecto secundario que le dice cuánta memoria se está usando actualmente:

```
gc()
#>           used (Mb) gc trigger (Mb) max used  (Mb)
#> Ncells 1031677 55.1    2073078  111  1823744  97.4
#> Vcells 5409972 41.3   15017895  115 15009387 114.6
```

`lobstr::mem_used()` es un envoltorio alrededor de `gc()` que imprime el número total de bytes utilizados:

```
mem_used()
#> 101.03 MB
```

Este número no coincidirá con la cantidad de memoria informada por su sistema operativo. Hay tres razones:

1. Incluye objetos creados por R pero no por el intérprete de R.
2. Tanto R como el sistema operativo son perezosos: no reclamarán memoria hasta que realmente se necesite. R podría estar reteniendo la memoria porque el sistema operativo aún no la ha solicitado.

3. R cuenta la memoria ocupada por objetos, pero puede haber espacios vacíos debido a objetos eliminados. Este problema se conoce como fragmentación de la memoria.

2.7. Respuestas de la prueba

1. Debe citar nombres no sintácticos con acentos graves: `` ` ``: por ejemplo, las variables `1`, `2` y `3`.

```
df <- data.frame(runif(3), runif(3))
names(df) <- c(1, 2)

df$`3` <- df$`1` + df$`2`
```

2. Ocupa unos 8 MB.

```
x <- runif(1e6)
y <- list(x, x, x)
obj_size(y)
#> 8.00 MB
```

3. `a` se copia cuando se modifica `b`, `b[[1]] <- 10`.

3. Vectores

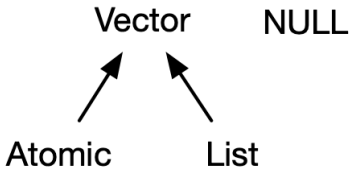
3.1. Introducción

Este capítulo analiza la familia más importante de tipos de datos en base R: vectores[1]. Si bien es probable que ya haya usado muchos (si no todos) de los diferentes tipos de vectores, es posible que no haya pensado profundamente en cómo están interrelacionados. En este capítulo, no cubriré los tipos de vectores individuales con demasiado detalle, pero le mostraré cómo encajan todos los tipos como un todo. Si necesita más detalles, puede encontrarlos en la documentación de R.

Los vectores vienen en dos sabores: vectores atómicos y listas [2]. Se diferencian en cuanto a los tipos de sus elementos: para los vectores atómicos, todos los elementos deben tener el mismo tipo; para las listas, los elementos pueden tener diferentes tipos. Si bien no es un vector, `NULL` está estrechamente relacionado con los vectores y, a menudo, cumple la función de un vector genérico de longitud cero. Este diagrama, que ampliaremos a lo largo de este capítulo, ilustra las relaciones básicas:

[1] En conjunto, todos los demás tipos de datos se conocen como tipos de "nodo", que incluyen cosas como funciones y entornos. Lo más probable es que te encuentres con este término altamente técnico cuando uses `gc()`: la "N" en `Ncells` significa nodos y la "V" en `Vcells` significa vectores.

[2] Algunos lugares en la documentación de R llaman a listas de vectores genéricos para enfatizar su diferencia con los vectores atómicos.



Cada vector también puede tener **atributos**, que puede considerar como una lista con nombre de metadatos arbitrarios. Dos atributos son particularmente importantes. El atributo **dimensión** convierte los vectores en matrices y arreglos y el atributo **clase** impulsa el sistema de objetos S3. Si bien aprenderá a usar S3 en el Chapter 13, aquí aprenderá sobre algunos de los vectores S3 más importantes: factores, fecha y hora, data frames y tibbles. Y aunque las estructuras 2D como matrices y data frames no son necesariamente lo que le viene a la mente cuando piensa en vectores, también aprenderá por qué R los considera vectores.

Prueba

Responda este breve cuestionario para determinar si necesita leer este capítulo. Si las respuestas le vienen a la mente rápidamente, puede saltarse cómodamente este capítulo. Puede comprobar sus respuestas en la Section 3.8.

1. ¿Cuáles son los cuatro tipos comunes de vectores atómicos? ¿Cuáles son los dos tipos raros?

2. ¿Qué son los atributos? ¿Cómo los consigues y los configuras?

3. ¿En qué se diferencia una lista de un vector atómico? ¿En qué se diferencia una matriz de un data frame?

4. ¿Puedes tener una lista que sea una matriz? ¿Puede un data frame tener una columna que sea una matriz?

5. ¿En qué se diferencian los tibbles de los data frames?

Estructura

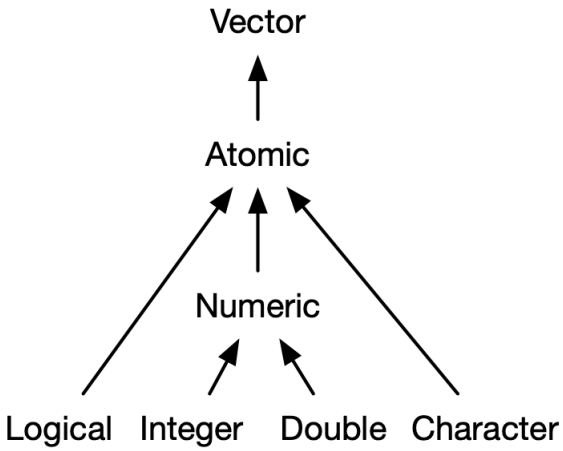
- La Section 3.2 te introduce a los vectores atómicos: lógico, entero, doble y de carácter. Estas son las estructuras de datos más simples de R.
- La Section 3.3 toma un pequeño desvío para discutir los atributos, la especificación de metadatos flexibles de R. Los atributos más importantes son los nombres, las dimensiones y la clase.
- La Section 3.4 analiza los tipos de vectores importantes que se construyen combinando vectores atómicos con atributos especiales. Estos incluyen factores, fechas, fechas-horas y duraciones.
- La Section 3.5 se sumerge en las listas. Las listas son muy similares a los vectores atómicos, pero tienen una diferencia clave: un elemento de una lista puede ser cualquier tipo de datos, incluida otra lista. Esto los hace adecuados para representar datos jerárquicos.
- La Section 3.6 te enseña sobre data frames y tibbles, que se utilizan para representar datos rectangulares. Combinan el comportamiento de listas y matrices para crear una estructura ideal para las necesidades de los datos estadísticos.

3.2. Vectores atómicos

Hay cuatro tipos principales de vectores atómicos: lógico, entero, doble y carácter (que contiene cadenas). En conjunto, los vectores enteros y dobles se conocen como vectores numéricos [3]. Hay dos tipos raros: complejos

[3] Esta es una ligera simplificación ya que R no usa "numérico" de manera consistente, a lo que volveremos en la Section 12.3.1.

y crudos. No los discutiré más porque los números complejos rara vez se necesitan en las estadísticas, y los vectores sin procesar son un tipo especial que solo se necesita cuando se manejan datos binarios.



3.2.1. Escalares

Cada uno de los cuatro tipos principales tiene una sintaxis especial para crear un valor individual, también conocido como **escalar**[4]:

- Los lógicos se pueden escribir completos (`TRUE` o `FALSE`) o abreviados (`T` o `F`).
- Los dobles se pueden especificar en formato decimal (`0.1234`), científico (`1.23e4`) o hexadecimal (`0xcafe`). Hay tres valores especiales únicos para los dobles: `Inf`, `-Inf` y `NaN` (no es un número). Estos son valores especiales definidos por el estándar de punto flotante.

[4]Técnicamente, el lenguaje R no posee escalares. Todo lo que parece un escalar es en realidad un vector de longitud uno. Esta es principalmente una distinción teórica, pero significa que expresiones como `1[1]` funcionan.

- Los enteros se escriben de forma similar a los dobles, pero deben ir seguidos de `L`[5] (`1234L`, `1e4L` o `0xcafeL`), y no pueden contener valores fraccionarios.

- Las cadenas están rodeadas por `"` (`"hola"`) o `'` (`'adiós'`). Los caracteres especiales se escapan con `\`; consulte `?Quotes` para obtener detalles completos.

3.2.2. Crear vectores más largos con `c()`

Para crear vectores más largos a partir de otros más cortos, use `c()`, abreviatura de combinar:

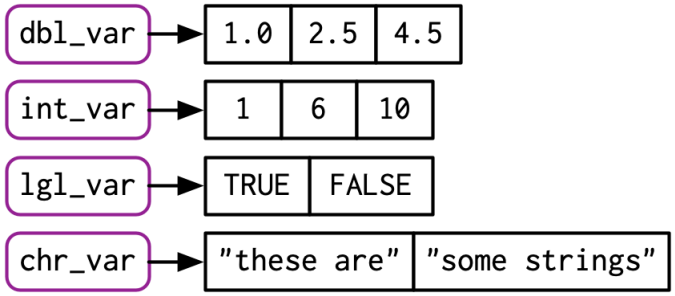
```
lgl_var <- c(TRUE, FALSE)
int_var <- c(1L, 6L, 10L)
dbl_var <- c(1, 2.5, 4.5)
chr_var <- c("these are", "some strings")
```

Cuando las entradas son vectores atómicos, `c()` siempre crea otro vector atómico; es decir, se aplana:

```
c(c(1, 2), c(3, 4))
#> [1] 1 2 3 4
```

En los diagramas, representaré los vectores como rectángulos conectados, por lo que el código anterior podría dibujarse de la siguiente manera:

[5] `L` no es intuitivo, y puede que te preguntes de dónde viene. En el momento en que se agregó `L` a R, el tipo de entero de R era equivalente a un entero largo en C, y el código C podía usar un sufijo de `l` o `L` para obligar a un número a ser un entero largo. Se decidió que `l` era demasiado similar visualmente a `i` (usado para números complejos en R), dejando `L`.



Puedes determinar el tipo de un vector con `typeof()`[6] y su longitud con `length()`.

```
typeof(lgl_var)
#> [1] "logical"
typeof(int_var)
#> [1] "integer"
typeof(dbl_var)
#> [1] "double"
typeof(chr_var)
#> [1] "character"
```

3.2.3. Valores Faltantes

R representa valores faltantes o desconocidos, con un valor centinela especial: `NA` (abreviatura de no aplicable). Los valores faltantes tienden a ser infecciosos: la mayoría de los cálculos que involucran un valor faltante devolverán otro valor faltante.

[6] Es posible que haya oído hablar de las funciones `mode()` y `storage.mode()` relacionadas. No los uses: existen solo por compatibilidad con S.

```
NA > 5
#> [1] NA
10 * NA
#> [1] NA
!NA
#> [1] NA
```

Sólo hay unas pocas excepciones a esta regla. Estos ocurren cuando alguna identidad se mantiene para todas las entradas posibles:

```
NA ^ 0
#> [1] 1
NA | TRUE
#> [1] TRUE
NA & FALSE
#> [1] FALSE
```

La propagación de faltantes conduce a un error común al determinar qué valores faltan en un vector:

```
x <- c(NA, 5, NA, 10)
x == NA
#> [1] NA NA NA NA
```

Este resultado es correcto (aunque un poco sorprendente) porque no hay motivo para creer que un valor faltante tiene el mismo valor que otro. En su lugar, use `is.na()` para probar la presencia de ausencias:

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

NB: Técnicamente, hay cuatro valores faltantes[7], uno para cada uno de los tipos atómicos: `NA` (lógico), `NA_integer_` (entero), `NA_real_` (doble) y `NA_character_` (carácter). Esta distinción generalmente no es importante porque 'NA' será forzado automáticamente al tipo correcto cuando sea necesario.

3.2.4. Pruebas y coerción

En general, puede **probar** si un vector es de un tipo dado con una función `is.*()`, pero estas funciones deben usarse con cuidado. `is.logical()`, `is.integer()`, `is.double()` y `is.character()` hacen lo que cabría esperar: prueban si un vector es un carácter, doble, entero o lógico. Evite `is.vector()`, `is.atomic()` y `is.numeric()`: no comprueban si tiene un vector, un vector atómico o un vector numérico; deberá leer detenidamente la documentación para descubrir qué es lo que realmente hacen.

Para los vectores atómicos, el tipo es una propiedad de todo el vector: todos los elementos deben ser del mismo tipo. Cuando intente combinar diferentes tipos, se **coaccionarán** en un orden fijo: carácter → doble → entero → lógico. Por ejemplo, la combinación de un carácter y un número entero produce un carácter:

```
str(c("a", 1))
#>  chr [1:2] "a" "1"
```

La coerción a menudo ocurre automáticamente. La mayoría de las funciones matemáticas (`+`, `log`, `abs`, etc.) se convertirán en numéricas. Esta coerción es particularmente útil para vectores lógicos porque `TRUE` se convierte en 1 y `FALSE` se convierte en 0.

[7] Bueno, técnicamente, cinco si incluimos `NA_complex_`, pero ignoramos este tipo "raro", como se mencionó anteriormente.

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
#> [1] 0 0 1

# Número total de VERDADEROS
sum(x)
#> [1] 1

# Proporción que son VERDADERAS
mean(x)
#> [1] 0.333
```

En general, puede forzar deliberadamente usando una función `as.*()`, como `as.logical()`, `as.integer()`, `as.double()` o `as.character()`. La coerción fallida de cadenas genera una advertencia y un valor faltante:

```
as.integer(c("1", "1.5", "a"))
#> Warning: NAs introduced by coercion
#> [1]  1  1 NA
```

3.2.5. Ejercicios

1. ¿Cómo se crean escalares crudos y complejos? (Ver `?raw` y `?complex`.)

2. Pon a prueba tu conocimiento de las reglas de coerción de vectores prediciendo el resultado de los siguientes usos de `c()`:

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

3. ¿Por qué `1 == "1"` es verdadero? ¿Por qué `-1 < FALSE` es verdadero? ¿Por qué `"one" < 2` es falso?

4. ¿Por qué el valor faltante predeterminado, `NA`, es un vector lógico? ¿Qué tienen de especial los vectores lógicos? (Pista: piensa en `c(FALSE, NA_character_)`.)

5. Precisamente, ¿qué prueban `is.atomic()`, `is.numeric()` y `is.vector()`?

3.3. Atributos

Es posible que haya notado que el conjunto de vectores atómicos no incluye una serie de estructuras de datos importantes como matrices, arreglos, factores o fechas y horas. Estos tipos se construyen sobre vectores atómicos agregando atributos. En esta sección, aprenderá los conceptos básicos de los atributos y cómo el atributo dim crea matrices y arreglos. En la siguiente sección, aprenderá cómo se usa el atributo de clase para crear vectores de S3, incluidos factores, fechas y fechas y horas.

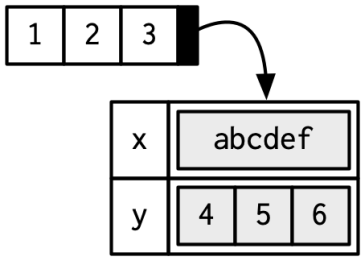
3.3.1. Conseguir y configurar

Puede pensar en los atributos como pares de nombre-valor [8] que adjuntan metadatos a un objeto. Los atributos individuales pueden recuperarse y modificarse con `attr()`, o recuperarse en masa con `attributes()`, y establecerse en masa con `structure()`.

```
a <- 1:3
attr(a, "x") <- "abcdef"
attr(a, "x")
#> [1] "abcdef"
```

[8] Los atributos se comportan como listas con nombre, pero en realidad son listas de pares. Las listas de pares son funcionalmente indistinguibles de las listas, pero son profundamente diferentes bajo el capó. Aprenderá más sobre ellos en la Section 18.6.1.

```
attr(a, "y") <- 4:6
str(attributes(a))
#> List of 2
#>  $ x: chr "abcdef"
#>  $ y: int [1:3] 4 5 6

# Or equivalently
a <- structure(
  1:3,
  x = "abcdef",
  y = 4:6
)
str(attributes(a))
#> List of 2
#>  $ x: chr "abcdef"
#>  $ y: int [1:3] 4 5 6
```



En general, los atributos deben considerarse efímeros. Por ejemplo, la mayoría de los atributos se pierden en la mayoría de las operaciones:

```
attributes(a[1])
#> NULL
```

```
attributes(sum(a))
#> NULL
```

Solo hay dos atributos que se conservan de forma rutinaria:

- **names**, un vector de caracteres que da a cada elemento un nombre.
- **dim**, abreviatura de dimensiones, un vector entero, que se utiliza para convertir vectores en matrices o arreglos.

Para conservar otros atributos, deberá crear su propia clase S3, el tema del Chapter 13.

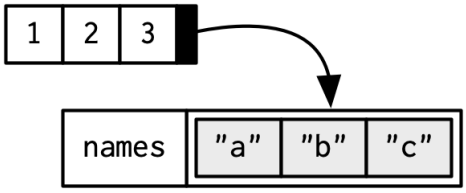
3.3.2. Nombres

Puede nombrar un vector de tres maneras:

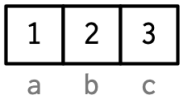
```
# Cuando lo crea:
x <- c(a = 1, b = 2, c = 3)

# Al asignar un vector de caracteres a names()
x <- 1:3
names(x) <- c("a", "b", "c")

# En línea con setNames():
x <- setNames(1:3, c("a", "b", "c"))
```

Evite usar `attr(x, "names")` ya que requiere escribir más y es menos legible que `names(x)`. Puede eliminar nombres de un vector usando `x <- unname(x)` o `names(x) <- NULL`.

Para ser técnicamente correcto, al dibujar el vector nombrado `x`, debería dibujarlo así:



Sin embargo, los nombres son tan especiales e importantes que, a menos que intente llamar la atención específicamente sobre la estructura de datos de los atributos, los usaré para etiquetar el vector directamente:



Para que sea útil con subconjuntos de caracteres (p. ej., la Section 4.5.1), los nombres deben ser únicos y no faltantes, pero R no impone esto. Dependiendo de cómo se establezcan los nombres, los nombres faltantes pueden ser `""` o `NA_character_`. Si faltan todos los nombres, `names()` devolverá `NULL`.

3.3.3. Dimensiones

Agregar un atributo `dim` a un vector le permite comportarse como una **matriz** bidimensional o una **matriz** multidimensional. Las matrices y los arreglos son principalmente herramientas matemáticas y estadísticas, no herramientas de programación, por lo que se usarán con poca frecuencia y solo se tratarán brevemente en este libro. Su característica más importante es el subconjunto multidimensional, que se trata en la Section 4.2.3.

Puede crear matrices y arreglos con `matrix()` y `array()`, o usando el formulario de asignación de `dim()`:

```
# Dos argumentos escalares especifican tamaños de fila y columna
x <- matrix(1:6, nrow = 2, ncol = 3)
x
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6

# Un argumento vectorial para describir todas las dimensiones
y <- array(1:12, c(2, 3, 2))
y
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
#> [2,]    8   10   12

# También puede modificar un objeto en su lugar configurando dim()
z <- 1:6
dim(z) <- c(3, 2)
z
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
```

Muchas de las funciones para trabajar con vectores tienen generalizaciones para matrices y arreglos:

Vector	Matriz	Arreglo
`names()`	`rownames()`, `colnames()`	`dimnames()`
`length()`	`nrow()`, `ncol()`	`dim()`
`c()`	`rbind()`, `cbind()`	`abind::abind()`
—	`t()`	`aperm()`
`is.null(dim(x))`	`is.matrix()`	`is.array()`

Un vector sin un conjunto de atributos "dim" a menudo se considera unidimensional, pero en realidad tiene dimensiones `NULL`. También puede tener matrices con una sola fila o una sola columna, o arreglos con una sola dimensión. Pueden imprimir de manera similar, pero se comportarán de manera diferente. Las diferencias no son demasiado importantes, pero es útil saber que existen en caso de que obtenga un resultado extraño de una función (`tapply()` es un infractor frecuente). Como siempre, usa `str()` para revelar las diferencias.

```
str(1:3)                   # 1d vector
#>  int [1:3] 1 2 3
str(matrix(1:3, ncol = 1)) # column vector
#>  int [1:3, 1] 1 2 3
str(matrix(1:3, nrow = 1)) # row vector
#>  int [1, 1:3] 1 2 3
str(array(1:3, 3))         # "array" vector
#>  int [1:3(1d)] 1 2 3
```

3.3.4. Ejercicios

1. ¿Cómo se implementa `setNames()`? ¿Cómo se implementa `unname()`? Lee el código fuente.

2. ¿Qué devuelve `dim()` cuando se aplica a un vector unidimensional? ¿Cuándo podría usar `NROW()` o `NCOL()`?

3. ¿Cómo describirías los siguientes tres objetos? ¿Qué los hace diferentes de `1:5`?

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

4. Un borrador inicial usó este código para ilustrar `structure()`:

```
structure(1:5, comment = "my attribute")
#> [1] 1 2 3 4 5
```

   Pero cuando imprime ese objeto, no ve el atributo de comentario. ¿Por qué? ¿Falta el atributo o hay algo más especial en él? (Sugerencia: intente usar la ayuda).

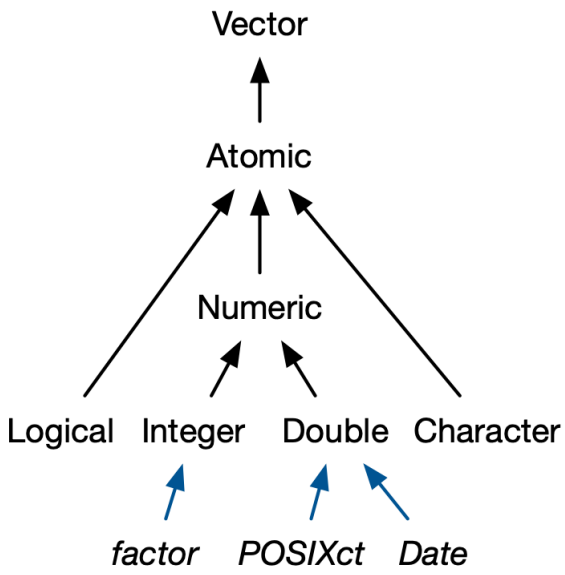
3.4. Vectores atómicos S3

Uno de los atributos vectoriales más importantes es la `clase`, que subyace en el sistema de objetos S3. Tener un atributo de clase convierte un objeto en un **objeto S3**, lo que significa que se comportará de manera diferente a un vector regular cuando se pasa a una función **genérica**. Cada objeto de S3 se crea sobre un tipo base y, a menudo, almacena información adicional en otros atributos. Aprenderá los detalles del sistema de objetos de S3 y cómo crear sus propias clases de S3 en el Chapter 13.

En esta sección, analizaremos cuatro vectores S3 importantes que se utilizan en la base R:

- Datos categóricos, donde los valores provienen de un conjunto fijo de niveles registrados en vectores de **factores**.
- Fechas (con resolución de día), que se registran en vectores **Fecha**.
- Fechas-horas (con resolución de segundos o subsegundos), que se almacenan en vectores **POSIXct**.

- Duraciones, que se almacenan en vectores **difftime**.

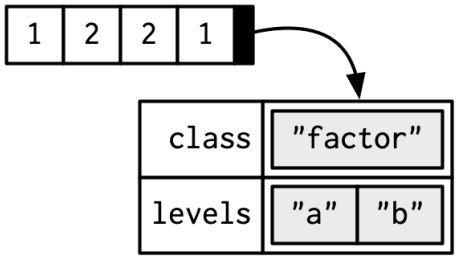


3.4.1. Factores

Un factor es un vector que solo puede contener valores predefinidos. Se utiliza para almacenar datos categóricos. Los factores se construyen sobre un vector entero con dos atributos: una `class`, "factor", que hace que se comporte de manera diferente a los vectores enteros normales, y `levels`, que define el conjunto de valores permitidos.

```
x <- factor(c("a", "b", "b", "a"))
x
#> [1] a b b a
#> Levels: a b
```

```
typeof(x)
#> [1] "integer"
attributes(x)
#> $levels
#> [1] "a" "b"
#>
#> $class
#> [1] "factor"
```



Los factores son útiles cuando conoce el conjunto de valores posibles, pero no todos están presentes en un conjunto de datos determinado. A diferencia de un vector de caracteres, cuando tabula un factor obtendrá recuentos de todas las categorías, incluso las no observadas:

```
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
#> sex_char
#> m
#> 3
table(sex_factor)
#> sex_factor
```

```
#> m f
#> 3 0
```

Los factores **ordenados** son una variación menor de los factores. En general, se comportan como factores regulares, pero el orden de los niveles es significativo (bajo, medio, alto) (una propiedad que algunas funciones de modelado y visualización aprovechan automáticamente).

```
grade <- ordered(c("b", "b", "a", "c"), levels = c("c", "b", "a"))
grade
#> [1] b b a c
#> Levels: c < b < a
```

En base R[9], tiende a encontrar factores con mucha frecuencia porque muchas funciones de base R (como `read.csv()` y `data.frame()`) convierten automáticamente los vectores de caracteres en factores. Esto es subóptimo porque no hay forma de que esas funciones conozcan el conjunto de todos los niveles posibles o su orden correcto: los niveles son una propiedad de la teoría o el diseño experimental, no de los datos. En su lugar, utilice el argumento `stringsAsFactors = FALSE` para suprimir este comportamiento y luego convierta manualmente los vectores de caracteres en factores usando su conocimiento de los datos "teóricos". Para conocer el contexto histórico de este comportamiento, recomiendo *stringsAsFactors: Una biografía no autorizada* de Roger Peng, y *stringsAsFactors = <sigh>* de Thomas Lumley.

Si bien los factores se ven (y a menudo se comportan como) vectores de caracteres, se construyen sobre números enteros. Así que tenga cuidado al tratarlos como cadenas. Algunos métodos de cadena (como `gsub()` y `grepl()`) forzarán automáticamente los factores a cadenas, otros (como `nchar()`) generarán un error y otros (como `c()`) usarán los valores enteros

[9] El tidyverse nunca fuerza automáticamente a los personajes a factores, y proporciona el paquete forcats (Wickham 2018) específicamente para trabajar con factores.

subyacentes. Por esta razón, normalmente es mejor convertir explícitamente los factores en vectores de caracteres si necesita un comportamiento similar al de una cadena.

3.4.2. Fechas

Los vectores de fecha se construyen sobre vectores dobles. Tienen clase "Date" y ningún otro atributo:

```
today <- Sys.Date()

typeof(today)
#> [1] "double"
attributes(today)
#> $class
#> [1] "Date"
```

El valor del doble (que se puede ver quitando la clase), representa el número de días desde 1970-01-01[10]:

```
date <- as.Date("1970-02-01")
unclass(date)
#> [1] 31
```

3.4.3. Fecha-Hora

Base R[11] proporciona dos formas de almacenar información de fecha y hora, POSIXct y POSIXlt. Estos son nombres ciertamente extraños: "POSIX"

[10] Esta fecha especial se conoce como la Época Unix.

[11] tidyverse proporciona el paquete lubridate (Grolemund and Wickham 2011) para trabajar con fechas y horas. Proporciona una serie de útiles ayudantes que funcionan con el tipo POSIXct base.

es la abreviatura de Portable Operating System Interface, que es una familia de estándares multiplataforma. "ct" representa la hora del calendario (el tipo `time_t` en C), y "lt" la hora local (el tipo `struct tm` en C). Aquí nos centraremos en `POSIXct`, porque es el más simple, está construido sobre un vector atómico y es más apropiado para usar en tramas de datos. Los vectores POSIXct se construyen sobre vectores dobles, donde el valor representa la cantidad de segundos desde 1970-01-01.

```
now_ct <- as.POSIXct("2018-08-01 22:00", tz = "UTC")
now_ct
#> [1] "2018-08-01 22:00:00 UTC"

typeof(now_ct)
#> [1] "double"
attributes(now_ct)
#> $class
#> [1] "POSIXct" "POSIXt"
#>
#> $tzone
#> [1] "UTC"
```

El atributo `tzone` controla solo cómo se formatea la fecha y la hora; no controla el instante de tiempo representado por el vector. Tenga en cuenta que la hora no se imprime si es medianoche.

```
structure(now_ct, tzone = "Asia/Tokyo")
#> [1] "2018-08-02 07:00:00 JST"
structure(now_ct, tzone = "America/New_York")
#> [1] "2018-08-01 18:00:00 EDT"
structure(now_ct, tzone = "Australia/Lord_Howe")
#> [1] "2018-08-02 08:30:00 +1030"
structure(now_ct, tzone = "Europe/Paris")
#> [1] "2018-08-02 CEST"
```

3.4.4. Duraciones

Las duraciones, que representan la cantidad de tiempo entre pares de fechas o fechas-horas, se almacenan en difftimes. Los tiempos de diferencia se construyen sobre los dobles y tienen un atributo de `unidades` que determina cómo se debe interpretar el número entero:

```
one_week_1 <- as.difftime(1, units = "weeks")
one_week_1
#> Time difference of 1 weeks

typeof(one_week_1)
#> [1] "double"
attributes(one_week_1)
#> $class
#> [1] "difftime"
#>
#> $units
#> [1] "weeks"

one_week_2 <- as.difftime(7, units = "days")
one_week_2
#> Time difference of 7 days

typeof(one_week_2)
#> [1] "double"
attributes(one_week_2)
#> $class
#> [1] "difftime"
#>
#> $units
#> [1] "days"
```

3.4.5. Ejercicios

1. ¿Qué tipo de objeto devuelve `table()`? ¿Cuál es su tipo? ¿Qué atributos tiene? ¿Cómo cambia la dimensionalidad a medida que tabula más variables?

2. ¿Qué le sucede a un factor cuando modificas sus niveles?

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

3. ¿Qué hace este código? ¿En qué se diferencian `f2` y `f3` de `f1`?

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

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

3.5. Listas

Las listas son un paso más en complejidad que los vectores atómicos: cada elemento puede ser de cualquier tipo, no solo vectores. Técnicamente hablando, cada elemento de una lista es en realidad del mismo tipo porque, como viste en la Section 2.3.3, cada elemento es realmente una *referencia* a otro objeto, que puede ser de cualquier tipo.

3.5.1. Creando

Construyes listas con `list()`:

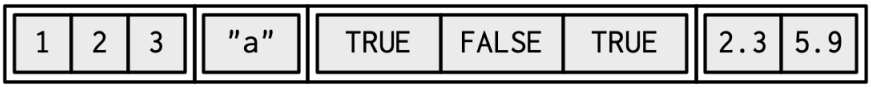
```
l1 <- list(
  1:3,
  "a",
  c(TRUE, FALSE, TRUE),
```

```
  c(2.3, 5.9)
)

typeof(l1)
#> [1] "list"

str(l1)
#> List of 4
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

Dado que los elementos de una lista son referencias, crear una lista no implica copiar los componentes en la lista. Por esta razón, el tamaño total de una lista puede ser más pequeño de lo esperado.

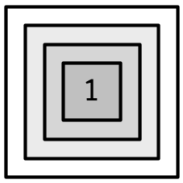
```
lobstr::obj_size(mtcars)
#> 7.21 kB

l2 <- list(mtcars, mtcars, mtcars, mtcars)
lobstr::obj_size(l2)
#> 7.29 kB
```

Las listas pueden contener objetos complejos, por lo que no es posible elegir un único estilo visual que funcione para todas las listas. En general, dibujaré listas como vectores, usando colores para recordarle la jerarquía.



Las listas a veces se denominan vectores **recursivos** porque una lista puede contener otras listas. Esto los hace fundamentalmente diferentes de los vectores atómicos.

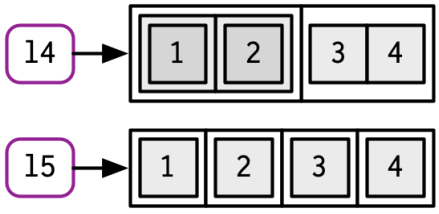
```
l3 <- list(list(list(1)))
str(l3)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : num 1
```



`c()` combinará varias listas en una sola. Si se le da una combinación de vectores atómicos y listas, `c()` convertirá los vectores en listas antes de combinarlos. Compara los resultados de `list()` y `c()`:

```
l4 <- list(list(1, 2), c(3, 4))
l5 <- c(list(1, 2), c(3, 4))
str(l4)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
str(l5)
#> List of 4
#>  $ : num 1
```

```
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```



3.5.2. Pruebas y coerción

El `typeof()` una lista es `list`. Puede probar una lista con `is.list()` y obligar a una lista con `as.list()`.

```
list(1:3)
#> [[1]]
#> [1] 1 2 3
as.list(1:3)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

Puedes convertir una lista en un vector atómico con `unlist()`. Las reglas para el tipo resultante son complejas, no están bien documentadas y no siempre son equivalentes a lo que obtendría con `c()`.

3.5.3. Matrices y arreglos

Con vectores atómicos, el atributo de dimensión se usa comúnmente para crear matrices. Con las listas, el atributo de dimensión se puede usar para crear matrices de listas o arreglos de listas:

```
l <- list(1:3, "a", TRUE, 1.0)
dim(l) <- c(2, 2)
l
#>      [,1]      [,2]
#> [1,] integer,3 TRUE
#> [2,] "a"       1

l[[1, 1]]
#> [1] 1 2 3
```

Estas estructuras de datos son relativamente esotéricas, pero pueden ser útiles si desea organizar objetos en una estructura similar a una cuadrícula. Por ejemplo, si está ejecutando modelos en una cuadrícula espacio-temporal, podría ser más intuitivo almacenar los modelos en una matriz 3D que coincida con la estructura de la cuadrícula.

3.5.4. Ejercicios

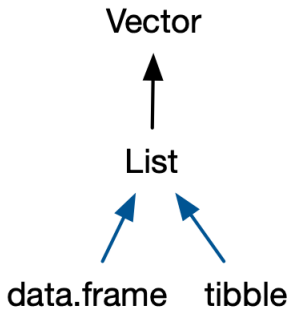
1. Enumere todas las formas en que una lista difiere de un vector atómico.

2. ¿Por qué necesita usar `unlist()` para convertir una lista en un vector atómico? ¿Por qué no funciona `as.vector()`?

3. Compare y contraste `c()` y `unlist()` al combinar una fecha y una fecha y hora en un solo vector.

3.6. Data frames y tibbles

Los dos vectores S3 más importantes construidos sobre las listas son los data frames y los tibbles.



Si realiza análisis de datos en R, utilizará data frames. Un data frame es una lista con nombre de vectores con atributos para (columna) `nombres`, `fila.nombres` [12], y su clase, "data.frame":

```
df1 <- data.frame(x = 1:3, y = letters[1:3])
typeof(df1)
#> [1] "list"
```

[12]Los nombres de fila son una de las estructuras de datos más sorprendentemente complejas en R. También han sido una fuente persistente de problemas de rendimiento a lo largo de los años. La implementación más sencilla es un vector de caracteres o entero, con un elemento para cada fila. Pero también hay una representación compacta para nombres de fila "automáticos" (enteros consecutivos), creada por `.set_row_names()`. R 3.5 tiene una forma especial de diferir la conversión de enteros a caracteres que está diseñada específicamente para acelerar `lm()`; consulte https://svn.r-project.org/R/branches/ALTREP/ALTREP.html#deferred_string_conversions para obtener detalles.

```
attributes(df1)
#> $names
#> [1] "x" "y"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#> [1] 1 2 3
```

A diferencia de una lista regular, un data frame tiene una restricción adicional: la longitud de cada uno de sus vectores debe ser la misma. Esto le da a los data frames su estructura rectangular y explica por qué comparten las propiedades de las matrices y las listas:

- Un data frame tiene `rownames()`[13] y `colnames()`. Los `names()` de un data frame son los nombres de las columnas.
- Un data frame tiene filas `nrow()` y columnas `ncol()`. La `longitud()` de un data frame da el número de columnas.

Los data frames son una de las ideas más grandes e importantes de R, y una de las cosas que diferencian a R de otros lenguajes de programación. Sin embargo, en los más de 20 años desde su creación, las formas en que las personas usan R han cambiado y algunas de las decisiones de diseño que tenían sentido en el momento en que se crearon los data frames ahora causan frustración.

Esta frustración condujo a la creación del tibble (Müller and Wickham 2018), una reinvención moderna del data frame. Los Tibbles están diseñados para ser (en la medida de lo posible) reemplazos directos de los data frames que solucionan esas frustraciones. Una forma concisa y divertida

[13] técnicamente, se recomienda usar `row.names()`, no `rownames()` con data frames, pero esta distinción rara vez es importante.

de resumir las principales diferencias es que los tibbles son vagos y hoscos: hacen menos y se quejan más. Verá lo que eso significa a medida que avance en esta sección.

Los tibbles son proporcionados por el paquete tibble y comparten la misma estructura que los data frames. La única diferencia es que el vector de clase es más largo e incluye `tbl_df`. Esto permite que los tibbles se comporten de manera diferente en las formas clave que discutiremos a continuación.

```
library(tibble)

df2 <- tibble(x = 1:3, y = letters[1:3])
typeof(df2)
#> [1] "list"

attributes(df2)
#> $class
#> [1] "tbl_df"     "tbl"        "data.frame"
#>
#> $row.names
#> [1] 1 2 3
#>
#> $names
#> [1] "x" "y"
```

3.6.1. Creando

Creas un data frame proporcionando pares de vector de nombre a `data.frame()`:

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c")
```

```
)
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

Si está utilizando una versión de R anterior a la 4.0.0, tenga cuidado con la conversión predeterminada de cadenas a factores. Use `stringsAsFactors = FALSE` para suprimir esto y mantener los vectores de caracteres como vectores de caracteres:

```
df1 <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE
)
str(df1)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

Crear un tibble es similar a crear un data frame. La diferencia entre los dos es que los tibbles nunca coaccionan su entrada (esta es una característica que los hace perezosos):

```
df2 <- tibble(
  x = 1:3,
  y = c("a", "b", "c")
)
str(df2)
#> tibble [3 × 2] (S3: tbl_df/tbl/data.frame)
#>  $ x: int [1:3] 1 2 3
#>  $ y: chr [1:3] "a" "b" "c"
```

Además, mientras que los data frames transforman automáticamente los nombres no sintácticos (a menos que `check.names = FALSE`), los tibbles no lo hacen (aunque sí imprimen nombres no sintácticos rodeados por `).

```
names(data.frame(`1` = 1))
#> [1] "X1"

names(tibble(`1` = 1))
#> [1] "1"
```

Si bien cada elemento de un data frame (o tibble) debe tener la misma longitud, tanto `data.frame()` como `tibble()` reciclarán entradas más cortas. Sin embargo, mientras que los data frames reciclan automáticamente las columnas que son múltiplos enteros de la columna más larga, los tibbles solo reciclan vectores de longitud uno.

```
data.frame(x = 1:4, y = 1:2)
#>   x y
#> 1 1 1
#> 2 2 2
#> 3 3 1
#> 4 4 2
data.frame(x = 1:4, y = 1:3)
#> Error in data.frame(x = 1:4, y = 1:3): arguments imply differing number o

tibble(x = 1:4, y = 1)
#> # A tibble: 4 × 2
#>       x     y
#>   <int> <dbl>
#> 1     1     1
#> 2     2     1
#> 3     3     1
```

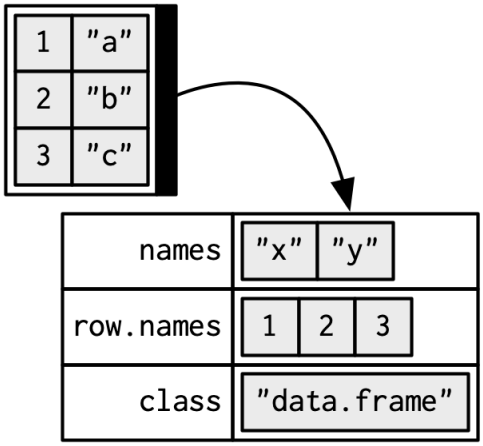
```
#> 4     4     1
tibble(x = 1:4, y = 1:2)
#> Error in `tibble()`:
#> ! Tibble columns must have compatible sizes.
#> • Size 4: Existing data.
#> • Size 2: Column `y`.
#>   Only values of size one are recycled.
```

Hay una última diferencia: `tibble()` te permite referirte a las variables creadas durante la construcción:

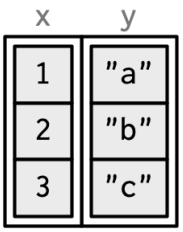
```
tibble(
  x = 1:3,
  y = x * 2
)
#> # A tibble: 3 × 2
#>       x     y
#>   <int> <dbl>
#> 1     1     2
#> 2     2     4
#> 3     3     6
```

(Las entradas se evalúan de izquierda a derecha).

Al dibujar data frames y tibbles, en lugar de centrarse en los detalles de implementación, es decir, los atributos:



Los dibujaré de la misma manera que una lista con nombre, pero los ordenaré para enfatizar su estructura en columnas.



3.6.2. Nombres de filas

Los data frames le permiten etiquetar cada fila con un nombre, un vector de caracteres que contiene solo valores únicos:

```
df3 <- data.frame(
  age = c(35, 27, 18),
  hair = c("blond", "brown", "black"),
  row.names = c("Bob", "Susan", "Sam")
)
df3
#>       age  hair
#> Bob    35 blond
#> Susan  27 brown
#> Sam    18 black
```

Puede obtener y establecer nombres de fila con `rownames()`, y puede usarlos para crear subconjuntos de filas:

```
rownames(df3)
#> [1] "Bob"   "Susan" "Sam"

df3["Bob", ]
#>     age  hair
#> Bob  35 blond
```

Los nombres de las filas surgen naturalmente si piensa en los data frames como estructuras 2D como matrices: las columnas (variables) tienen nombres, por lo que las filas (observaciones) también deberían tenerlos. La mayoría de las matrices son numéricas, por lo que es importante tener un lugar para almacenar etiquetas de caracteres. Pero esta analogía con las matrices es engañosa porque las matrices poseen una propiedad importante que los data frames no tienen: son transponibles. En las matrices, las filas y las columnas son intercambiables, y la transposición de una matriz da como resultado otra matriz (la transposición nuevamente da como resultado la matriz original). Sin embargo, con los data frames, las filas y las columnas no son intercambiables: la transposición de un data frame no es un data frame.

Hay tres razones por las que los nombres de las filas no son deseables:

- Los metadatos son datos, por lo que almacenarlos de manera diferente al resto de los datos es fundamentalmente una mala idea. También significa que necesita aprender un nuevo conjunto de herramientas para trabajar con nombres de fila; no puede usar lo que ya sabe sobre la manipulación de columnas.
- Los nombres de fila son una abstracción deficiente para etiquetar filas porque solo funcionan cuando una fila se puede identificar con una sola cadena. Esto falla en muchos casos, por ejemplo, cuando desea identificar una fila por un vector que no es un carácter (por ejemplo, un punto de tiempo), o con múltiples vectores (por ejemplo, posición, codificado por latitud y longitud).
- Los nombres de las filas deben ser únicos, por lo que cualquier duplicación de filas (por ejemplo, de arranque) creará nuevos nombres de fila. Si desea hacer coincidir las filas de antes y después de la transformación, deberá realizar una complicada cirugía de hilo.

```
df3[c(1, 1, 1), ]
#>       age  hair
#> Bob    35 blond
#> Bob.1  35 blond
#> Bob.2  35 blond
```

Por estas razones, los tibbles no admiten nombres de fila. En cambio, el paquete tibble proporciona herramientas para convertir fácilmente los nombres de las filas en una columna regular con `rownames_to_column()`, o el argumento `rownames` en `as_tibble()`:

```
as_tibble(df3, rownames = "name")
#> # A tibble: 3 × 3
#>   name    age hair
#>   <chr> <dbl> <chr>
```

```
#> 1 Bob      35 blond
#> 2 Susan    27 brown
#> 3 Sam      18 black
```

3.6.3. Imprimir

Una de las diferencias más obvias entre tibbles y data frames es cómo se imprimen. Supongo que ya está familiarizado con la forma en que se imprimen los data frames, por lo que aquí resaltaré algunas de las mayores diferencias utilizando un conjunto de datos de ejemplo incluido en el paquete dplyr:

```
dplyr::starwars
#> # A tibble: 87 × 14
#>    name       height  mass hair_color skin_color eye_color birth_year
#>    <chr>       <int> <dbl> <chr>      <chr>      <chr>          <dbl>
#>  1 Luke Skyw…    172    77 blond      fair       blue            19
#>  2 C-3PO         167    75 <NA>       gold       yellow         112
#>  3 R2-D2          96    32 <NA>       white, bl… red             33
#>  4 Darth Vad…    202   136 none       white      yellow          41.9
#>  5 Leia Orga…    150    49 brown      light      brown           19
#>  6 Owen Lars     178   120 brown, gr… light      blue            52
#>  7 Beru Whit…    165    75 brown      light      blue            47
#>  8 R5-D4          97    32 <NA>       white, red red             NA
#>  9 Biggs Dar…    183    84 black      light      brown           24
#> 10 Obi-Wan K…    182    77 auburn, w… fair       blue-gray       57
#> #   77 more rows
#> #   7 more variables: sex <chr>, gender <chr>, homeworld <chr>,
#> #   species <chr>, films <list>, vehicles <list>, starships <list>
```

- Tibbles solo muestra las primeras 10 filas y todas las columnas que caben en la pantalla. Las columnas adicionales se muestran en la parte inferior.

- Cada columna está etiquetada con su tipo, abreviado en tres o cuatro letras.
- Las columnas anchas se truncan para evitar que una sola cadena larga ocupe una fila completa. (Esto todavía es un trabajo en progreso: es un compromiso complicado entre mostrar tantas columnas como sea posible y mostrar las columnas en su totalidad).
- Cuando se usa en entornos de consola que lo admiten, el color se usa juiciosamente para resaltar información importante y restar énfasis a los detalles complementarios.

3.6.4. Subconjunto

Como aprenderá en el Chapter 4, puede crear un subconjunto de un data frame o un tibble como una estructura 1D (donde se comporta como una lista), o una estructura 2D (donde se comporta como una matriz).

En mi opinión, los data frames tienen dos comportamientos de subconjunto no deseados:

- Cuando crea subconjuntos de columnas con `df[, vars]`, obtendrá un vector si `vars` selecciona una variable; de lo contrario, obtendrá un data frame. Esta es una fuente frecuente de errores cuando se usa `[` en una función, a menos que siempre recuerde usar `df[, vars, drop = FALSE]`.
- Cuando intenta extraer una sola columna con `df$x` y no hay ninguna columna `x`, un data frame seleccionará cualquier variable que comience con `x`. Si ninguna variable comienza con `x`, `df$x` devolverá `NULL`. Esto facilita seleccionar la variable incorrecta o seleccionar una variable que no existe[14].

[14]Podemos hacer que R advierta sobre este comportamiento (llamado coincidencia parcial) configurando `options(warnPartialMatchDollar = TRUE)`.

Tibbles modifica estos comportamientos para que un `[` siempre devuelva un tibble, y un `$` no haga coincidencias parciales y advierta si no puede encontrar una variable (esto es lo que hace que Tibbles sea hosco).

```
df1 <- data.frame(xyz = "a")
df2 <- tibble(xyz = "a")

str(df1$x)
#>  chr "a"
str(df2$x)
#> Warning: Unknown or uninitialised column: `x`.
#>  NULL
```

La insistencia de un tibble en devolver un data frame de `[` puede causar problemas con el código heredado, que a menudo usa `df[, "col"]` para extraer una sola columna. Si desea una sola columna, le recomiendo usar `df[["col"]]`. Esto comunica claramente su intención y funciona tanto con data frames como con tibbles.

3.6.5. Pruebas y coacción

Para verificar si un objeto es un data frame o tibble, use `is.data.frame()`:

```
is.data.frame(df1)
#> [1] TRUE
is.data.frame(df2)
#> [1] TRUE
```

Por lo general, no debería importar si tiene un tibble o un data frame, pero si necesita estar seguro, use `is_tibble()`:

```
is_tibble(df1)
#> [1] FALSE
is_tibble(df2)
#> [1] TRUE
```

Puede convertir un objeto en un data frame con `as.data.frame()` o en un tibble con `as_tibble()`.

3.6.6. Columnas de lista

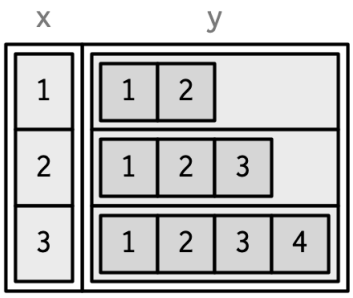
Dado que un data frame es una lista de vectores, es posible que un data frame tenga una columna que sea una lista. Esto es muy útil porque una lista puede contener cualquier otro objeto: esto significa que puede colocar cualquier objeto en un data frame. Esto le permite mantener los objetos relacionados juntos en una fila, sin importar cuán complejos sean los objetos individuales. Puede ver una aplicación de esto en el capítulo "Muchos modelos" de *R para la Ciencia de Datos*, http://r4ds.had.co.nz/many-models.html.

Las columnas de lista están permitidas en data frames, pero debe hacer un poco de trabajo adicional agregando la columna de lista después de la creación o envolviendo la lista en `I()`[15].

```
df <- data.frame(x = 1:3)
df$y <- list(1:2, 1:3, 1:4)

data.frame(
  x = 1:3,
  y = I(list(1:2, 1:3, 1:4))
)
```

[15] `I()` es la abreviatura de identidad y se usa a menudo para indicar que una entrada debe dejarse como está y no transformarse automáticamente.

```
#>   x          y
#> 1 1       1, 2
#> 2 2    1, 2, 3
#> 3 3 1, 2, 3, 4
```



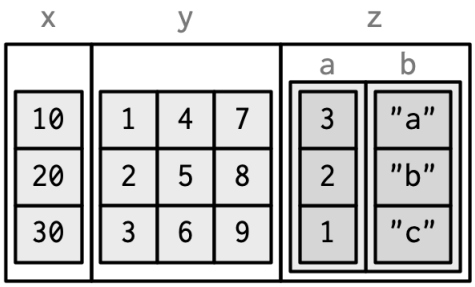
Las columnas de lista son más fáciles de usar con tibbles porque se pueden incluir directamente dentro de `tibble()` y se imprimirán ordenadamente:

```
tibble(
  x = 1:3,
  y = list(1:2, 1:3, 1:4)
)
#> # A tibble: 3 × 2
#>       x y
#>   <int> <list>
#> 1     1 <int [2]>
#> 2     2 <int [3]>
#> 3     3 <int [4]>
```

3.6.7. Columnas de matriz y data frame

Siempre que el número de filas coincida con el data frame, también es posible tener una matriz o arreglo como columna de un data frame. (Esto requiere una ligera extensión a nuestra definición de data frame: no es la longitud, `length()`, de cada columna lo que debe ser igual, sino el `NROW()`.) En cuanto a las columnas de lista, debe agregarlas después de la creación o envolverlas en `I()`.

```
dfm <- data.frame(
  x = 1:3 * 10
)
dfm$y <- matrix(1:9, nrow = 3)
dfm$z <- data.frame(a = 3:1, b = letters[1:3], stringsAsFactors = FALSE)

str(dfm)
#> 'data.frame':    3 obs. of  3 variables:
#>  $ x: num  10 20 30
#>  $ y: int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
#>  $ z:'data.frame':   3 obs. of  2 variables:
#>   ..$ a: int  3 2 1
#>   ..$ b: chr  "a" "b" "c"
```



Las columnas de matrices y data frames requieren un poco de precaución. Muchas funciones que trabajan con data frames asumen que todas las columnas son vectores. Además, la pantalla impresa puede resultar confusa.

```
dfm[1, ]
#>    x y.1 y.2 y.3 z.a z.b
#> 1 10   1   4   7   3   a
```

3.6.8. Ejercicios

1. ¿Puede tener un data frame con cero filas? ¿Qué pasa con las columnas cero?

2. ¿Qué sucede si intenta establecer nombres de fila que no son únicos?

3. Si `df` es un data frame, ¿qué puede decir acerca de `t(df)` y `t(t(df))`? Realice algunos experimentos, asegurándose de probar diferentes tipos de columnas.

4. ¿Qué hace `as.matrix()` cuando se aplica a un data frame con columnas de diferentes tipos? ¿En qué se diferencia de `data.matrix()`?

3.7. `NULL`

Para terminar este capítulo, quiero hablar sobre una última estructura de datos importante que está estrechamente relacionada con los vectores: `NULL`. `NULL` es especial porque tiene un tipo único, siempre tiene una longitud cero y no puede tener ningún atributo:

```
typeof(NULL)
#> [1] "NULL"

length(NULL)
#> [1] 0

x <- NULL
attr(x, "y") <- 1
#> Error in attr(x, "y") <- 1: attempt to set an attribute on NULL
```

Puedes probar `NULL`s con `is.null()`:

```
is.null(NULL)
#> [1] TRUE
```

Hay dos usos comunes de `NULL`:

- Representar un vector vacío (un vector de longitud cero) de tipo arbitrario. Por ejemplo, si usas `c()` pero no incluyes ningún argumento, obtienes `NULL`, y concatenar `NULL` a un vector lo dejará sin cambios:

  ```
  c()
  #> NULL
  ```

- Para representar un vector ausente. Por ejemplo, `NULL` a menudo se usa como argumento de función predeterminado, cuando el argumento es opcional pero el valor predeterminado requiere algún cálculo (consulte la Section 6.5.3 para obtener más información al respecto). Contraste esto con `NA` que se usa para indicar que un *elemento* de un vector está ausente.

Si está familiarizado con SQL, conocerá el `NULL` relacional y puede esperar que sea igual que el de R. Sin embargo, la base de datos `NULL` es en realidad equivalente a `NA` de R.

3.8. Respuestas de la prueba

1. Los cuatro tipos comunes de vectores atómicos son lógicos, enteros, dobles y de carácter. Los dos tipos más raros son complejos y crudos.

2. Los atributos le permiten asociar metadatos adicionales arbitrarios a cualquier objeto. Puede obtener y establecer atributos individuales con `attr(x, "y")` y `attr(x, "y") <- value`; o puede obtener y establecer todos los atributos a la vez con `attributes()`.

3. Los elementos de una lista pueden ser de cualquier tipo (incluso una lista); los elementos de un vector atómico son todos del mismo tipo. De manera similar, todos los elementos de una matriz deben ser del mismo tipo; en un data frame, diferentes columnas pueden tener diferentes tipos.

4. Puede crear una matriz de lista asignando dimensiones a una lista. Puede convertir una matriz en una columna de un data frame con `df$x <- matrix()`, o usando `I()` al crear un nuevo data frame `data.frame(x = I(matrix()))`.

5. Los Tibbles tienen un método de impresión mejorado, nunca obligan a las cadenas a factores y proporcionan métodos de subconjunto más estrictos.

4. Subconjunto

4.1. Introducción

Los operadores de subconjuntos de R son rápidos y potentes. Dominarlos le permite realizar operaciones complejas de manera sucinta de una manera que pocos otros idiomas pueden igualar. La creación de subconjuntos en R es fácil de aprender pero difícil de dominar porque necesita interiorizar una serie de conceptos interrelacionados:

- Hay seis formas de crear subconjuntos de vectores atómicos.
- Hay tres operadores de subconjuntos, `[[`, `[` y `$`.
- Los operadores de creación de subconjuntos interactúan de manera diferente con diferentes tipos de vectores (por ejemplo, vectores atómicos, listas, factores, matrices y data frames).
- La creación de subconjuntos se puede combinar con la asignación.

La creación de subconjuntos es un complemento natural de `str()`. Mientras que `str()` le muestra todas las piezas de cualquier objeto (su estructura), la creación de subconjuntos le permite extraer las piezas que le interesan. Para objetos grandes y complejos, recomiendo usar el RStudio Viewer interactivo, que puedes activar con `View(my_object)`.

Prueba

Responda este breve cuestionario para determinar si necesita leer este capítulo. Si las respuestas le vienen a la mente rápidamente, puede saltarse cómodamente este capítulo. Comprueba tus respuestas en la Section 4.6.

1. ¿Cuál es el resultado de subdividir un vector con enteros positivos, enteros negativos, un vector lógico o un vector de caracteres?
2. ¿Cuál es la diferencia entre `[`, `[[` y `$` cuando se aplica a una lista?
3. ¿Cuándo debería usar `drop = FALSE`?
4. Si `x` es una matriz, ¿qué hace `x[] <- 0`? ¿En qué se diferencia de `x <- 0`?
5. ¿Cómo puede usar un vector con nombre para volver a etiquetar variables categóricas?

Estructura

- La Section 4.2 comienza enseñándote acerca de `[`. Aprenderá las seis formas de crear subconjuntos de vectores atómicos. Luego aprenderá cómo actúan esas seis formas cuando se usan para crear subconjuntos de listas, matrices y data frames.
- La Section 4.3 amplía su conocimiento de los operadores de subconjuntos para incluir `[[` y `$` y se centra en los principios importantes de simplificar frente a preservar.
- La Section 4.4 aprenderá el arte de la subasignación, que combina subconjuntos y asignación para modificar partes de un objeto.
- La Section 4.5 llo guía a través de ocho aplicaciones importantes, pero no obvias, de subconjuntos para resolver problemas que a menudo encuentra en el análisis de datos.

4.2. Selección de varios elementos

Utilice `[` para seleccionar cualquier número de elementos de un vector. Para ilustrar, aplicaré `[` a vectores atómicos 1D, y luego mostraré cómo esto se generaliza a objetos más complejos y más dimensiones.

4.2.1. Vectores atómicos

Exploremos los diferentes tipos de subconjuntos con un vector simple, `x`.

```
x <- c(2.1, 4.2, 3.3, 5.4)
```

Tenga en cuenta que el número después del punto decimal representa la posición original en el vector.

Hay seis cosas que puede usar para crear subconjuntos de un vector:

- **Los enteros positivos** devuelven elementos en las posiciones especificadas:

```
x[c(3, 1)]
#> [1] 3.3 2.1
x[order(x)]
#> [1] 2.1 3.3 4.2 5.4

# Los índices duplicados duplicarán los valores
x[c(1, 1)]
#> [1] 2.1 2.1

# Los números reales se truncan silenciosamente a enteros
x[c(2.1, 2.9)]
#> [1] 4.2 4.2
```

- **Los enteros negativos** excluyen elementos en las posiciones especificadas:

```
x[-c(3, 1)]
#> [1] 4.2 5.4
```

Tenga en cuenta que no puede mezclar números enteros positivos y negativos en un solo subconjunto:

```
x[c(-1, 2)]
#> Error in x[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

- **Los vectores lógicos** seleccionan elementos donde el valor lógico correspondiente es `TRUE`. Este es probablemente el tipo de subconjunto más útil porque puede escribir una expresión que usa un vector lógico:

```
x[c(TRUE, TRUE, FALSE, FALSE)]
#> [1] 2.1 4.2
x[x > 3]
#> [1] 4.2 3.3 5.4
```

En `x[y]`, ¿qué sucede si `x` e `y` tienen longitudes diferentes? El comportamiento está controlado por las **reglas de reciclaje**, donde el más corto de los dos se recicla al largo del más largo. Esto es conveniente y fácil de entender cuando uno de `x` e `y` tiene la longitud uno, pero recomiendo evitar el reciclaje para otras longitudes porque las reglas se aplican de manera inconsistente en la base R.

```
x[c(TRUE, FALSE)]
#> [1] 2.1 3.3
# Equivalent to
x[c(TRUE, FALSE, TRUE, FALSE)]
#> [1] 2.1 3.3
```

Tenga en cuenta que un valor faltante en el índice siempre produce un valor faltante en la salida:

```
x[c(TRUE, TRUE, NA, FALSE)]
#> [1] 2.1 4.2  NA
```

- **Nada** devuelve el vector original. Esto no es útil para vectores 1D, pero, como verá en breve, es muy útil para matrices, data frames y arreglos. También puede ser útil junto con la asignación.

```
x[]
#> [1] 2.1 4.2 3.3 5.4
```

- **Zero** devuelve un vector de longitud cero. Esto no es algo que normalmente haga a propósito, pero puede ser útil para generar datos de prueba.

```
x[0]
#> numeric(0)
```

- Si el vector tiene nombre, también puede usar **vectores de caracteres** para devolver elementos con nombres coincidentes.

```
(y <- setNames(x, letters[1:4]))
#>   a   b   c   d
#> 2.1 4.2 3.3 5.4
y[c("d", "c", "a")]
#>   d   c   a
#> 5.4 3.3 2.1

# Al igual que los índices enteros, puede repetir índices
y[c("a", "a", "a")]
#>   a   a   a
#> 2.1 2.1 2.1

# Al crear un subconjunto con [, los nombres siempre coinciden exactamente
z <- c(abc = 1, def = 2)
z[c("a", "d")]
```

```
#> <NA> <NA>
#>   NA   NA
```

NB: Los factores no se tratan de manera especial cuando se subdividen. Esto significa que la creación de subconjuntos utilizará el vector entero subyacente, no los niveles de caracteres. Esto suele ser inesperado, por lo que debe evitar subconjuntos con factores:

```
y[factor("b")]
#>   a
#> 2.1
```

4.2.2. Listas

La creación de subconjuntos de una lista funciona de la misma manera que la creación de subconjuntos de un vector atómico. Usar `[` siempre devuelve una lista; `[[` y `$`, como se describe en la Section 4.3, le permiten extraer elementos de una lista.

4.2.3. Matrices y arreglos

Puede crear subconjuntos de estructuras de dimensiones superiores de tres maneras:

- Con múltiples vectores.
- Con un solo vector.
- Con una matriz.

La forma más común de crear subconjuntos de matrices (2D) y arreglos (>2D) es una generalización simple de subconjuntos 1D: proporcione un índice 1D para cada dimensión, separados por una coma. El subconjunto en blanco ahora es útil porque le permite mantener todas las filas o todas las columnas.

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- c("A", "B", "C")
a[1:2, ]
#>      A B C
#> [1,] 1 4 7
#> [2,] 2 5 8
a[c(TRUE, FALSE, TRUE), c("B", "A")]
#>      B A
#> [1,] 4 1
#> [2,] 6 3
a[0, -2]
#>      A C
```

Por defecto, `[` simplifica los resultados a la dimensionalidad más baja posible. Por ejemplo, las dos expresiones siguientes devuelven vectores 1D. Aprenderá cómo evitar "reducir" el número de dimensiones en la Section 4.2.5:

```
a[1, ]
#> A B C
#> 1 4 7
a[1, 1]
#> A
#> 1
```

Debido a que tanto las matrices como los arreglos son solo vectores con atributos especiales, puede crear subconjuntos con un solo vector, como si fueran un vector 1D. Tenga en cuenta que las matrices en R se almacenan en orden de columna principal:

```
vals <- outer(1:5, 1:5, FUN = "paste", sep = ",")
vals
#>      [,1]  [,2]  [,3]  [,4]  [,5]
```

```
#> [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
#> [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
#> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
#> [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
#> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"

vals[c(4, 15)]
#> [1] "4,1" "5,3"
```

También puede crear subconjuntos de estructuras de datos de mayor dimensión con una matriz de enteros (o, si se nombra, una matriz de caracteres). Cada fila de la matriz especifica la ubicación de un valor y cada columna corresponde a una dimensión de la matriz. Esto significa que puede usar una matriz de 2 columnas para crear un subconjunto de una matriz, una matriz de 3 columnas para crear un subconjunto de una matriz 3D, etc. El resultado es un vector de valores:

```
select <- matrix(ncol = 2, byrow = TRUE, c(
  1, 1,
  3, 1,
  2, 4
))
vals[select]
#> [1] "1,1" "3,1" "2,4"
```

4.2.4. Data frames y tibbles

Data frames tienen las características tanto de listas como de matrices:

- Cuando se subdividen con un solo índice, se comportan como listas e indexan las columnas, por lo que `df[1:2]` selecciona las dos primeras columnas.

- Al crear subconjuntos con dos índices, se comportan como matrices, por lo que `df[1:3, ]` selecciona las primeras tres *filas* (y todas las columnas)[1].

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])

df[df$x == 2, ]
#>   x y z
#> 2 2 2 b
df[c(1, 3), ]
#>   x y z
#> 1 1 3 a
#> 3 3 1 c

# Hay dos formas de seleccionar columnas de un data frame
# Como una lista
df[c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
# Como una matriz
df[, c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c

# Hay una diferencia importante si selecciona una sola
# columna: el subconjunto de la matriz se simplifica de forma predeterminada, el
```

[1] Si viene de Python, es probable que esto sea confuso, ya que probablemente esperaría que `df[1:3, 1:2]` seleccione tres columnas y dos filas. Generalmente, R "piensa" en las dimensiones en términos de filas y columnas, mientras que Python lo hace en términos de columnas y filas.

```
# subconjunto de la lista no
str(df["x"])
#> 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
str(df[, "x"])
#>  int [1:3] 1 2 3
```

Subdividir un tibble con `[` siempre devuelve un tibble:

```
df <- tibble::tibble(x = 1:3, y = 3:1, z = letters[1:3])

str(df["x"])
#> tibble [3 × 1] (S3: tbl_df/tbl/data.frame)
#>  $ x: int [1:3] 1 2 3
str(df[, "x"])
#> tibble [3 × 1] (S3: tbl_df/tbl/data.frame)
#>  $ x: int [1:3] 1 2 3
```

4.2.5. Preservando la dimensionalidad

De forma predeterminada, subdividir una matriz o data frame con un solo número, un solo nombre o un vector lógico que contenga un solo `TRUE` simplificará la salida devuelta, es decir, devolverá un objeto con menor dimensionalidad. Para conservar la dimensionalidad original, debe usar `drop = FALSE`.

- Para matrices y arreglos, se eliminarán todas las dimensiones con longitud 1:

  ```
  a <- matrix(1:4, nrow = 2)
  str(a[1, ])
  #>  int [1:2] 1 3
  ```

```
str(a[1, , drop = FALSE])
#>  int [1, 1:2] 1 3
```

- Data frames con una sola columna devolverán solo el contenido de esa columna:

```
df <- data.frame(a = 1:2, b = 1:2)
str(df[, "a"])
#>  int [1:2] 1 2

str(df[, "a", drop = FALSE])
#> 'data.frame':    2 obs. of  1 variable:
#>  $ a: int  1 2
```

El comportamiento predeterminado `drop = TRUE` es una fuente común de errores en las funciones: verifica su código con un data frame o matriz con varias columnas, y funciona. Seis meses después, usted (u otra persona) lo usa con un data frame de una sola columna y falla con un error desconcertante. Al escribir funciones, acostúmbrese a usar siempre `drop = FALSE` al subdividir un objeto 2D. Por esta razón, los tibbles tienen por defecto `drop = FALSE`, y `[` siempre devuelve otro tibble.

El subconjunto de factores también tiene un argumento `drop`, pero su significado es bastante diferente. Controla si se conservan o no los niveles (en lugar de las dimensiones), y su valor predeterminado es `FALSE`. Si encuentra que está usando `drop = TRUE` mucho, a menudo es una señal de que debería estar usando un vector de caracteres en lugar de un factor.

```
z <- factor(c("a", "b"))
z[1]
#> [1] a
#> Levels: a b
z[1, drop = TRUE]
#> [1] a
#> Levels: a
```

4.2.6. Ejercicios

1. Solucione cada uno de los siguientes errores comunes de creación de subconjuntos de data frames:

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

2. ¿Por qué el siguiente código arroja cinco valores faltantes? (Pista: ¿por qué es diferente de `x[NA_real_]`?)

```
x <- 1:5
x[NA]
#> [1] NA NA NA NA NA
```

3. ¿Qué devuelve `upper.tri()`? ¿Cómo funciona el subconjunto de una matriz con él? ¿Necesitamos reglas adicionales de creación de subconjuntos para describir su comportamiento?

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

4. ¿Por qué `mtcars[1:20]` devuelve un error? ¿En qué se diferencia de los `mtcars[1:20, ]` similares?

5. Implemente su propia función que extraiga las entradas diagonales de una matriz (debería comportarse como `diag(x)` donde `x` es una matriz).

6. ¿Qué hace `df[is.na(df)] <- 0`? ¿Como funciona?

4.3. Selección de un solo elemento

Hay otros dos operadores de subconjuntos: `[[` y `$`. `[[` se usa para extraer elementos individuales, mientras que `x$y` es una abreviatura útil para `x[["y"]]`.

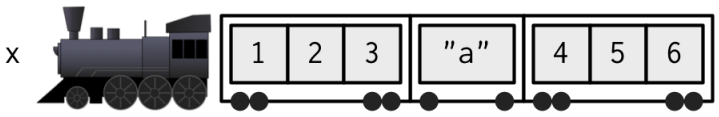
4.3.1. `[[`

`[[` es más importante cuando se trabaja con listas porque subdividir una lista con `[` siempre devuelve una lista más pequeña. Para ayudar a que esto sea más fácil de entender, podemos usar una metáfora:

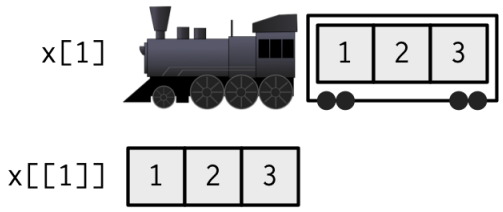
> Si la lista `x` es un tren que transporta objetos, entonces `x[[5]]` es el objeto en el vagón 5; `x[4:6]` es un tren de vagones 4-6.
>
> — @RLangTip, https://twitter.com/RLangTip/status/268375867468681216

Usemos esta metáfora para hacer una lista simple:

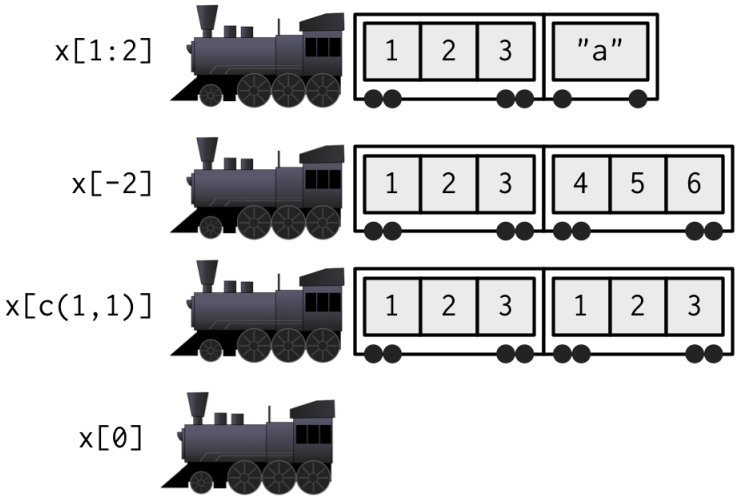
```
x <- list(1:3, "a", 4:6)
```



Al extraer un solo elemento, tiene dos opciones: puede crear un tren más pequeño, es decir, menos vagones, o puede extraer el contenido de un vagón en particular. Esta es la diferencia entre `[` y `[[`:



Al extraer elementos múltiples (¡o incluso cero!), debe hacer un tren más pequeño:



Debido a que `[[` solo puede devolver un solo elemento, debe usarlo con un solo entero positivo o una sola cadena. Si usa un vector con `[[`, se subdividirá recursivamente, es decir, `x[[c(1, 2)]]` es equivalente a `x[[1]][[2]]`. Esta es una característica peculiar que pocos conocen, así que recomiendo evitarla en favor de `purrr::pluck()`, sobre la cual aprenderá en la Section 4.3.3.

Si bien debe usar `[[` cuando trabaja con listas, también recomendaría usarlo con vectores atómicos siempre que desee extraer un solo valor. Por ejemplo, en lugar de escribir:

```
for (i in 2:length(x)) {
  out[i] <- fun(x[i], out[i - 1])
}
```

Es mejor escribir:

```
for (i in 2:length(x)) {
  out[[i]] <- fun(x[[i]], out[[i - 1]])
}
```

Si lo hace, refuerza la expectativa de que está recibiendo y estableciendo valores individuales.

4.3.2. `$`

`$` es un operador abreviado: `x$y` es aproximadamente equivalente a `x[["y", exact = FALSE]]`. A menudo se usa para acceder a variables en un data frame, como en `mtcars$cyl` o `diamonds$carat`. Un error común con `$` es usarlo cuando tienes el nombre de una columna almacenada en una variable:

```
var <- "cyl"
# No funciona - mtcars$var traducido a mtcars[["var"]]
mtcars$var
#> NULL

# En su lugar use [[
mtcars[[var]]
#>  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

La única diferencia importante entre `$` y `[[` es que `$` automáticamente hace (de izquierda a derecha) coincidencias parciales sin previo aviso:

```
x <- list(abc = 1)
x$a
#> [1] 1
x[["a"]]
#> NULL
```

Para ayudar a evitar este comportamiento, recomiendo configurar la opción global `warnPartialMatchDollar` en `TRUE`:

```
options(warnPartialMatchDollar = TRUE)
x$a
#> Warning in x$a: partial match of 'a' to 'abc'
#> [1] 1
```

(Para data frames, también puede evitar este problema usando tibbles, que nunca hacen coincidencias parciales.)

4.3.3. Índices faltantes y fuera de los límites

Es útil comprender lo que sucede con `[[` cuando usa un índice "no válido". La siguiente tabla resume lo que sucede cuando crea un subconjunto de un vector lógico, una lista y un `NULL` con un objeto de longitud cero (como `NULL` o `logical()`), valores fuera de los límites (OOB) o un valor faltante (por ejemplo, `NA_integer_`) con `[[`. Cada celda muestra el resultado de dividir la estructura de datos nombrada en la fila por el tipo de índice descrito en la columna. Solo he mostrado los resultados para vectores lógicos, pero otros vectores atómicos se comportan de manera similar, devolviendo elementos del mismo tipo (NB: int = entero; chr = carácter).

`row[[col]]`	Longitud cero	OOB (int)	OOB (chr)	Faltante
Atómico	Error	Error	Error	Error
Lista	Error	Error	`NULL`	`NULL`
`NULL`	`NULL`	`NULL`	`NULL`	`NULL`

Si se nombra el vector que se indexa, los nombres de los componentes OOB, faltantes o `NULL` serán `<NA>`.

Las inconsistencias en la tabla anterior llevaron al desarrollo de `purrr::pluck()` y `purrr::chuck()`. Cuando falta el elemento, `pluck()` siempre devuelve `NULL` (o el valor del argumento `.default`) y `chuck()` siempre arroja un error. El comportamiento de `pluck()` lo hace ideal para la indexación en estructuras de datos profundamente anidadas donde el componente que desea puede no existir (como es común cuando se trabaja con datos JSON de API web). `pluck()` también te permite mezclar índices enteros y de caracteres, y proporciona un valor predeterminado alternativo si un elemento no existe:

```
x <- list(
  a = list(1, 2, 3),
  b = list(3, 4, 5)
)

purrr::pluck(x, "a", 1)
#> [1] 1

purrr::pluck(x, "c", 1)
#> NULL

purrr::pluck(x, "c", 1, .default = NA)
#> [1] NA
```

4.3.4. `@` y `slot()`

Hay dos operadores de subconjuntos adicionales, que son necesarios para los objetos de S4: `@` (equivalente a `$`) y `slot()` (equivalente a `[[`). `@` es más restrictivo que `$` ya que devolverá un error si la ranura no existe. Estos se describen con más detalle en el Chapter 15.

4.3.5. Ejercicios

1. Piense en tantas formas como sea posible para extraer el tercer valor de la variable `cyl` en el conjunto de datos `mtcars`.

2. Dado un modelo lineal, por ejemplo, `mod <- lm(mpg ~ wt, data = mtcars)`, extraiga los grados de libertad residuales. Luego extraiga la R al cuadrado del resumen del modelo (`summary (mod)`)

4.4. Subconjunto y asignación

Todos los operadores de subconjunto se pueden combinar con la asignación para modificar los valores seleccionados de un vector de entrada: esto se denomina subasignación. La forma básica es `x[i] <- valor`:

```
x <- 1:5
x[c(1, 2)] <- c(101, 102)
x
#> [1] 101 102   3   4   5
```

Te recomiendo que te asegures de que `length(valor)` sea lo mismo que `length(x[i])`, y que `i` sea único. Esto se debe a que, si bien R reciclará si es necesario, esas reglas son complejas (particularmente si `i` contiene valores faltantes o duplicados) y pueden causar problemas.

Con las listas, puede usar `x[[i]] <- NULL` para eliminar un componente. Para agregar un literal `NULL`, use `x[i] <- list(NULL)`:

```
x <- list(a = 1, b = 2)
x[["b"]] <- NULL
str(x)
#> List of 1
#>  $ a: num 1

y <- list(a = 1, b = 2)
y["b"] <- list(NULL)
str(y)
#> List of 2
#>  $ a: num 1
#>  $ b: NULL
```

La creación de subconjuntos sin nada puede ser útil con la asignación porque conserva la estructura del objeto original. Compara las siguientes dos expresiones. En el primero, `mtcars` sigue siendo un data frame porque solo está cambiando el contenido de `mtcars`, no `mtcars` en sí. En el segundo, `mtcars` se convierte en una lista porque está cambiando el objeto al que está vinculado.

```
mtcars[] <- lapply(mtcars, as.integer)
is.data.frame(mtcars)
#> [1] TRUE

mtcars <- lapply(mtcars, as.integer)
is.data.frame(mtcars)
#> [1] FALSE
```

4.5. Aplicaciones

Los principios descritos anteriormente tienen una amplia variedad de aplicaciones útiles. A continuación se describen algunos de los más importantes. Si bien muchos de los principios básicos de creación de subconjuntos ya se han incorporado en funciones como `subset()`, `merge()` y `dplyr::arrange()`, será valioso comprender mejor cómo se han implementado esos principios. cuando se encuentra con situaciones en las que las funciones que necesita no existen.

4.5.1. Tablas de búsqueda (subconjunto de caracteres)

La coincidencia de caracteres es una forma poderosa de crear tablas de búsqueda. Digamos que quieres convertir abreviaturas:

```
x <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "Male", f = "Female", u = NA)
lookup[x]
#>        m        f        u        f        f        m        m
#>   "Male" "Female"       NA "Female" "Female"   "Male"   "Male"
```

Tenga en cuenta que si no quiere nombres en el resultado, use `unname()` para eliminarlos.

```
unname(lookup[x])
#> [1] "Male"   "Female" NA       "Female" "Female" "Male"   "Male"
```

4.5.2. Coincidencia y fusión a mano (subconjunto de enteros)

También puede tener tablas de búsqueda más complicadas con múltiples columnas de información. Por ejemplo, supongamos que tenemos un vector de grados enteros y una tabla que describe sus propiedades:

```
grades <- c(1, 2, 2, 3, 1)

info <- data.frame(
  grade = 3:1,
  desc = c("Excellent", "Good", "Poor"),
  fail = c(F, F, T)
)
```

Entonces, digamos que queremos duplicar la tabla `info` para que tengamos una fila para cada valor en `grade`. Una forma elegante de hacer esto es combinando `match()` y un subconjunto de enteros (`match(aguja, pajar)` devuelve la posición donde se encuentra cada `aguja` en el `pajar`).

```
id <- match(grades, info$grade)
id
#> [1] 3 2 2 1 3
info[id, ]
#>     grade      desc  fail
#> 3       1      Poor  TRUE
#> 2       2      Good FALSE
#> 2.1     2      Good FALSE
#> 1       3 Excellent FALSE
#> 3.1     1      Poor  TRUE
```

Si está haciendo coincidir varias columnas, primero deberá colapsarlas en una sola columna (con, por ejemplo, `interaction ()`). Sin embargo, normalmente es mejor cambiar a una función diseñada específicamente para unir varias tablas como `merge()` o `dplyr::left_join()`.

4.5.3. Muestras aleatorias y bootstraps (subconjunto de enteros)

Puede usar índices enteros para muestrear o arrancar aleatoriamente un vector o un data frame. Simplemente use `sample(n)` para generar una permutación aleatoria de `1:n`, y luego use los resultados para dividir los valores:

```
df <- data.frame(x = c(1, 2, 3, 1, 2), y = 5:1, z = letters[1:5])

# Reordenar aleatoriamente
df[sample(nrow(df)), ]
#>   x y z
#> 5 2 1 e
#> 3 3 3 c
#> 4 1 2 d
#> 1 1 5 a
#> 2 2 4 b

# Seleccionar aleatoriamente 3 filas
df[sample(nrow(df), 3), ]
#>   x y z
#> 4 1 2 d
#> 2 2 4 b
#> 1 1 5 a

# Seleccione 6 réplicas de arranque
df[sample(nrow(df), 6, replace = TRUE), ]
#>     x y z
#> 5   2 1 e
#> 5.1 2 1 e
#> 5.2 2 1 e
#> 2   2 4 b
```

```
#> 3   3 3 c
#> 3.1 3 3 c
```

Los argumentos de `sample()` controlan el número de muestras a extraer, y también si el muestreo se realiza con o sin reemplazo.

4.5.4. Ordenación (subconjunto de enteros)

`order()` toma un vector como entrada y devuelve un vector entero que describe cómo ordenar el vector dividido en subconjuntos[2]:

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

Para desempatar, puede proporcionar variables adicionales a `order()`. También puede cambiar el orden de ascendente a descendente utilizando `decreasing = TRUE`. De forma predeterminada, cualquier valor que falte se colocará al final del vector; sin embargo, puede eliminarlos con `na.last = NA` o ponerlos al frente con `na.last = FALSE`.

Para dos o más dimensiones, `order()` y el subconjunto de enteros facilita el orden de las filas o las columnas de un objeto:

```
# Reordenar al azar df
df2 <- df[sample(nrow(df)), 3:1]
df2
#>   z y x
```

[2]Estos son índices de "extracción", es decir, `order(x)[i]` es un índice de dónde se encuentra cada `x[i]`. No es un índice de dónde debe enviarse `x[i]`.

```
#> 5 e 1 2
#> 1 a 5 1
#> 4 d 2 1
#> 2 b 4 2
#> 3 c 3 3

df2[order(df2$x), ]
#>   z y x
#> 1 a 5 1
#> 4 d 2 1
#> 5 e 1 2
#> 2 b 4 2
#> 3 c 3 3
df2[, order(names(df2))]
#>   x y z
#> 5 2 1 e
#> 1 1 5 a
#> 4 1 2 d
#> 2 2 4 b
#> 3 3 3 c
```

Puede ordenar los vectores directamente con `sort()`, o de manera similar `dplyr::arrange()`, para ordenar un data frame.

4.5.5. Expansión de recuentos agregados (subconjunto de enteros)

A veces obtiene un data frame donde filas idénticas se han colapsado en una y se ha agregado una columna de conteo. `rep()` y el subconjunto de enteros hacen que sea fácil de descomprimir, porque podemos aprovechar la vectorización de `rep()`: `rep(x, y)` repite `x[i]` `y[i]` veces .

```
df <- data.frame(x = c(2, 4, 1), y = c(9, 11, 6), n = c(3, 5, 1))
rep(1:nrow(df), df$n)
#> [1] 1 1 1 2 2 2 2 2 3

df[rep(1:nrow(df), df$n), ]
#>     x  y n
#> 1   2  9 3
#> 1.1 2  9 3
#> 1.2 2  9 3
#> 2   4 11 5
#> 2.1 4 11 5
#> 2.2 4 11 5
#> 2.3 4 11 5
#> 2.4 4 11 5
#> 3   1  6 1
```

4.5.6. Eliminar columnas de data frames (caracteres subsetting)

Hay dos formas de eliminar columnas de un data frame. Puede establecer columnas individuales para `NULL`:

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df$z <- NULL
```

O puede extraer un subconjunto para devolver solo las columnas que desea:

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df[c("x", "y")]
#>   x y
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

Si solo conoce las columnas que no desea, use las operaciones de configuración para determinar qué columnas conservar:

```
df[setdiff(names(df), "z")]
#>   x y
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

4.5.7. Selección de filas en función de una condición (subconjunto lógico)

Debido a que el subconjunto lógico le permite combinar fácilmente condiciones de varias columnas, es probablemente la técnica más utilizada para extraer filas de un data frame.

```
mtcars[mtcars$gear == 5, ]
#>                 mpg cyl  disp  hp drat   wt qsec vs am gear carb
#> Porsche 914-2  26.0   4 120.3  91 4.43 2.14 16.7  0  1    5    2
#> Lotus Europa   30.4   4  95.1 113 3.77 1.51 16.9  1  1    5    2
#> Ford Pantera L 15.8   8 351.0 264 4.22 3.17 14.5  0  1    5    4
#> Ferrari Dino   19.7   6 145.0 175 3.62 2.77 15.5  0  1    5    6
#> Maserati Bora  15.0   8 301.0 335 3.54 3.57 14.6  0  1    5    8

mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
#>                mpg cyl  disp  hp drat   wt qsec vs am gear carb
#> Porsche 914-2 26.0   4 120.3  91 4.43 2.14 16.7  0  1    5    2
#> Lotus Europa  30.4   4  95.1 113 3.77 1.51 16.9  1  1    5    2
```

Recuerde utilizar los operadores booleanos vectoriales `&` y `|`, no los operadores escalares de cortocircuito `&&` y `||`, que son más útiles dentro de las sentencias if. Y no olvide las leyes de De Morgan, que pueden ser útiles para simplificar las negaciones:

- `!(X & Y)` es lo mismo que `!X | !Y`
- `!(X | Y)` es lo mismo que `!X & !Y`

Por ejemplo, `!(X & !(Y | Z))` se simplifica en `!X | !!(Y|Z)`, y luego a `!X | Y | Z`.

4.5.8. Álgebra booleana versus conjuntos (lógicos y enteros subsetting)

Es útil ser consciente de la equivalencia natural entre las operaciones con conjuntos (subconjuntos enteros) y el álgebra booleana (subconjuntos lógicos). El uso de operaciones de conjuntos es más efectivo cuando:

- Quieres encontrar el primero (o el último) `TRUE`.
- Tienes muy pocos `TRUE` y muchos `FALSE`; una representación establecida puede ser más rápida y requerir menos almacenamiento.

`which()` le permite convertir una representación booleana en una representación entera. No hay una operación inversa en la base R, pero podemos crear una fácilmente:

```
x <- sample(10) < 4
which(x)
#> [1] 2 3 4

unwhich <- function(x, n) {
  out <- rep_len(FALSE, n)
  out[x] <- TRUE
  out
}
unwhich(which(x), 10)
#>  [1] FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

4. Subconjunto

Creemos dos vectores lógicos y sus equivalentes enteros, y luego exploremos la relación entre las operaciones booleanas y de conjuntos.

```
(x1 <- 1:10 %% 2 == 0)
#>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
(x2 <- which(x1))
#> [1]  2  4  6  8 10
(y1 <- 1:10 %% 5 == 0)
#>  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
(y2 <- which(y1))
#> [1]  5 10

# X & Y <-> intersect(x, y)
x1 & y1
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
intersect(x2, y2)
#> [1] 10

# X | Y <-> union(x, y)
x1 | y1
#>  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
union(x2, y2)
#> [1]  2  4  6  8 10  5

# X & !Y <-> setdiff(x, y)
x1 & !y1
#>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
setdiff(x2, y2)
#> [1] 2 4 6 8

# xor(X, Y) <-> setdiff(union(x, y), intersect(x, y))
xor(x1, y1)
#>  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
setdiff(union(x2, y2), intersect(x2, y2))
```

```
#> [1] 2 4 6 8 5
```

Al aprender subconjuntos por primera vez, un error común es usar `x[which(y)]` en lugar de `x[y]`. Aquí, `which()` no logra nada: cambia de subconjunto lógico a entero, pero el resultado es exactamente el mismo. En casos más generales, hay dos diferencias importantes.

- Cuando el vector lógico contiene `NA`, el subconjunto lógico reemplaza estos valores con `NA` mientras que `which()` simplemente elimina estos valores. No es raro usar `which()` para este efecto secundario, pero no lo recomiendo: nada sobre el nombre "cuál" implica la eliminación de valores faltantes.

- `x[-which(y)]` **no** es equivalente a `x[!y]`: si `y` es todo FALSO, `which(y)` será `integer(0)` y `-integer(0)` sigue siendo `integer(0)`, por lo que no obtendrá valores, en lugar de todos los valores.

En general, evite cambiar de subconjunto lógico a entero a menos que desee, por ejemplo, el primer o último valor `TRUE`.

4.5.9. Ejercicios

1. ¿Cómo permutarías aleatoriamente las columnas de un data frame? (Esta es una técnica importante en los bosques aleatorios). ¿Puede permutar simultáneamente las filas y las columnas en un solo paso?

2. ¿Cómo seleccionaría una muestra aleatoria de `m` filas de un data frame? ¿Qué pasaría si la muestra tuviera que ser contigua (es decir, con una fila inicial, una fila final y todas las filas intermedias)?

3. ¿Cómo podría poner las columnas en un data frame en orden alfabético?

4.6. Respuestas de la

1. Los enteros positivos seleccionan elementos en posiciones específicas, los enteros negativos descartan elementos; los vectores lógicos mantienen los elementos en las posiciones correspondientes a `TRUE`; los vectores de caracteres seleccionan elementos con nombres coincidentes.

2. `[` selecciona sub-listas: siempre devuelve una lista. Si lo usa con un solo entero positivo, devuelve una lista de longitud uno. `[[` selecciona un elemento dentro de una lista. `$` es una abreviatura conveniente: `x$y` es equivalente a `x[["y"]]`.

3. Use `drop = FALSE` si está creando subconjuntos de una matriz, un arreglo o un data frame y desea conservar las dimensiones originales. Casi siempre deberías usarlo cuando hagas subconjuntos dentro de una función.

4. Si `x` es una matriz, `x[] <- 0` reemplazará cada elemento con 0, manteniendo el mismo número de filas y columnas. Por el contrario, `x <- 0` reemplaza completamente la matriz con el valor 0.

5. Un vector de caracteres con nombre puede actuar como una tabla de búsqueda simple: `c(x = 1, y = 2, z = 3)[c("y", "z", "x")]`

5. Flujo de control

5.1. Introducción

Hay dos herramientas principales de flujo de control: opciones y bucles. Las opciones, como declaraciones `if` y llamadas `switch()`, le permiten ejecutar código diferente dependiendo de la entrada. Los bucles, como `for` y `while`, le permiten ejecutar código repetidamente, normalmente con opciones cambiantes. Espero que ya esté familiarizado con los conceptos básicos de estas funciones, por lo que cubriré brevemente algunos detalles técnicos y luego presentaré algunas características útiles, pero menos conocidas.

El sistema de condiciones (mensajes, advertencias y errores), del que aprenderá en el Chapter 8, también proporciona un flujo de control no local.

Prueba

¿Quieres saltarte este capítulo? Anímate, si puedes responder las siguientes preguntas. Encuentre las respuestas al final del capítulo en la Section 5.4.

- ¿Cuál es la diferencia entre `if` y `ifelse()`?
- En el siguiente código, ¿cuál será el valor de `y` si `x` es `TRUE`? ¿Qué pasa si `x` es `FALSE`? ¿Qué pasa si `x` es `NA`?

```
y <- if (x) 3
```

- ¿Qué devuelve `switch("x", x = , y = 2, z = 3)`?

Estructura

- La Section 5.2 se sumerge en los detalles de `if`, luego analiza los parientes cercanos `ifelse()` y `switch()`.
- La Section 5.3 comienza recordándole la estructura básica del bucle for en R, analiza algunos errores comunes y luego habla sobre las declaraciones `while` y `repeat` relacionadas.

5.2. Opciones

La forma básica de una sentencia if en R es la siguiente:

```
if (condition) true_action
if (condition) true_action else false_action
```

Si `conditionn` es `TRUE`, se evalúa `true_action`; si `condition` es `FALSE`, se evalúa la `false_action` opcional.

Por lo general, las acciones son declaraciones compuestas contenidas dentro de `{`:

```
grade <- function(x) {
  if (x > 90) {
    "A"
  } else if (x > 80) {
    "B"
  } else if (x > 50) {
```

```
    "C"
  } else {
    "F"
  }
}
```

`if` devuelve un valor para que pueda asignar los resultados:

```
x1 <- if (TRUE) 1 else 2
x2 <- if (FALSE) 1 else 2

c(x1, x2)
#> [1] 1 2
```

(Recomiendo asignar los resultados de una declaración `if` solo cuando la expresión completa cabe en una línea; de lo contrario, tiende a ser difícil de leer.)

Cuando usa la forma de argumento único sin una declaración else, `if` invisiblemente (Section 6.7.2) devuelve `NULL` si la condición es `FALSE`. Dado que funciones como `c()` y `paste()` descartan entradas `NULL`, esto permite una expresión compacta de ciertos modismos:

```
greet <- function(name, birthday = FALSE) {
  paste0(
    "Hi ", name,
    if (birthday) " and HAPPY BIRTHDAY"
  )
}
greet("Maria", FALSE)
#> [1] "Hi Maria"
greet("Jaime", TRUE)
#> [1] "Hi Jaime and HAPPY BIRTHDAY"
```

5.2.1. Entradas inválidas

La condición debe evaluarse como un solo `TRUE` o `FALSE`. La mayoría de las otras entradas generarán un error:

```
if ("x") 1
#> Error in if ("x") 1: argument is not interpretable as logical
if (logical()) 1
#> Error in if (logical()) 1: argument is of length zero
if (NA) 1
#> Error in if (NA) 1: missing value where TRUE/FALSE needed
if (c(TRUE, FALSE)) 1
#> Error in if (c(TRUE, FALSE)) 1: the condition has length > 1
```

5.2.2. if vectorizado

Dado que `if` solo funciona con un solo `TRUE` o `FALSE`, es posible que se pregunte qué hacer si tiene un vector de valores lógicos. Manejar vectores de valores es el trabajo de `ifelse()`: una función vectorizada con vectores `test`, `sí` y `no` (que se reciclarán a la misma longitud):

```
x <- 1:10
ifelse(x %% 5 == 0, "XXX", as.character(x))
#>  [1] "1"   "2"   "3"   "4"   "XXX" "6"   "7"   "8"   "9"   "XXX"

ifelse(x %% 2 == 0, "even", "odd")
#>  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"
#> [10] "even"
```

Tenga en cuenta que los valores faltantes se propagarán a la salida.

Recomiendo usar `ifelse()` solo cuando los vectores `sí` y `no` son del mismo tipo, ya que de otro modo es difícil predecir el tipo de salida. Ver

https://vctrs.r-lib.org/articles/stability.html#ifelse para una discusión adicional.

Otro equivalente vectorizado es el más general `dplyr::case_when()`. Utiliza una sintaxis especial para permitir cualquier número de pares de vectores de condición:

```
dplyr::case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  is.na(x) ~ "???",
  TRUE ~ as.character(x)
)
#>  [1] "1"    "2"    "3"    "4"    "fizz" "6"    "buzz" "8"    "9"
#> [10] "fizz"
```

5.2.3. declaración `switch()`

Estrechamente relacionada con `if` está la sentencia `switch()`. Es un equivalente compacto de propósito especial que le permite reemplazar código como:

```
x_option <- function(x) {
  if (x == "a") {
    "option 1"
  } else if (x == "b") {
    "option 2"
  } else if (x == "c") {
    "option 3"
  } else {
    stop("Invalid `x` value")
```

```
  }
}
```

con la más sucinta:

```
x_option <- function(x) {
  switch(x,
    a = "option 1",
    b = "option 2",
    c = "option 3",
    stop("Invalid `x` value")
  )
}
```

El último componente de un `switch()` siempre debería arrojar un error; de lo contrario, las entradas no coincidentes devolverán invisiblemente `NULL`:

```
(switch("c", a = 1, b = 2))
#> NULL
```

Si varias entradas tienen la misma salida, puede dejar el lado derecho de `=` vacío y la entrada "caerá" al siguiente valor. Esto imita el comportamiento de la sentencia `switch` de C:

```
legs <- function(x) {
  switch(x,
    cow = ,
    horse = ,
    dog = 4,
    human = ,
    chicken = 2,
```

```
    plant = 0,
    stop("Unknown input")
  )
}
legs("cow")
#> [1] 4
legs("dog")
#> [1] 4
```

También es posible usar `switch()` con una `x` numérica, pero es más difícil de leer y tiene modos de falla no deseados si `x` no es un número entero. Recomiendo usar `switch()` solo con entradas de caracteres.

5.2.4. Ejercicios

1. ¿Qué tipo de vector devuelve cada una de las siguientes llamadas a `ifelse()`?

   ```
   ifelse(TRUE, 1, "no")
   ifelse(FALSE, 1, "no")
   ifelse(NA, 1, "no")
   ```

   Lee la documentación y escribe las reglas con tus propias palabras.

2. ¿Por qué funciona el siguiente código?

   ```
   x <- 1:10
   if (length(x)) "not empty" else "empty"
   #> [1] "not empty"

   x <- numeric()
   if (length(x)) "not empty" else "empty"
   #> [1] "empty"
   ```

5.3. Bucles

`for` los bucles se utilizan para iterar sobre los elementos de un vector. Tienen la siguiente forma básica:

```
for (item in vector) perform_action
```

Para cada elemento en `vector`, `perform_action` se llama una vez; actualizando el valor de `item` cada vez.

```
for (i in 1:3) {
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
```

(Al iterar sobre un vector de índices, es convencional usar nombres de variables muy cortos como `i`, `j`, or `k`.)

N.B.: `for` asigna el `item` al entorno actual, sobrescribiendo cualquier variable existente con el mismo nombre:

```
i <- 100
for (i in 1:3) {}
i
#> [1] 3
```

Hay dos formas de terminar un bucle `for` antes de tiempo:

- `next` sale de la iteración actual.
- `break` sale de todo el bucle `for`.

```
for (i in 1:10) {
  if (i < 3)
    next

  print(i)

  if (i >= 5)
    break
}
#> [1] 3
#> [1] 4
#> [1] 5
```

5.3.1. Errores comunes

Hay tres errores comunes a tener en cuenta al usar `for`. Primero, si está generando datos, asegúrese de asignar previamente el contenedor de salida. De lo contrario, el ciclo será muy lento; consulte las Secciones Section 23.2.2 y Section 24.6 para obtener más detalles. La función `vector()` es útil aquí.

```
means <- c(1, 50, 20)
out <- vector("list", length(means))
for (i in 1:length(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
```

A continuación, tenga cuidado con la iteración sobre `1:length(x)`, que fallará de manera inútil si `x` tiene una longitud de 0:

```
means <- c()
out <- vector("list", length(means))
for (i in 1:length(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
#> Error in rnorm(10, means[[i]]): invalid arguments
```

Esto ocurre porque `:` funciona tanto con secuencias crecientes como decrecientes:

```
1:length(means)
#> [1] 1 0
```

Utilice `seq_along(x)` en su lugar. Siempre devuelve un valor de la misma longitud que `x`:

```
seq_along(means)
#> integer(0)

out <- vector("list", length(means))
for (i in seq_along(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
```

Finalmente, es posible que encuentre problemas al iterar sobre los vectores de S3, ya que los bucles normalmente eliminan los atributos:

```
xs <- as.Date(c("2020-01-01", "2010-01-01"))
for (x in xs) {
  print(x)
}
#> [1] 18262
#> [1] 14610
```

Solucione esto llamando a `[[` usted mismo:

```
for (i in seq_along(xs)) {
  print(xs[[i]])
}
#> [1] "2020-01-01"
#> [1] "2010-01-01"
```

5.3.2. Herramientas relacionadas

Los bucles `for` son útiles si conoce de antemano el conjunto de valores que desea iterar. Si no lo sabe, hay dos herramientas relacionadas con especificaciones más flexibles:

- `while(condition) action`: ejecuta `action` mientras `condition` sea `TRUE`.
- `repeat(action)`: repite `action` siempre (i.e. hasta que encuentre `break`).

R no tiene un equivalente a la sintaxis `do {acción} while (condition)` que se encuentra en otros idiomas.

Puede reescribir cualquier bucle `for` para usar `while` en su lugar, y puede reescribir cualquier bucle `while` para usar `repeat`, pero lo contrario no es cierto. Eso significa que `while` es más flexible que `for`, y `repeat` es más flexible que `while`. Sin embargo, es una buena práctica usar la solución menos flexible a un problema, por lo que debe usar `for` siempre que sea posible.

En términos generales, no debería necesitar usar bucles `for` para tareas de análisis de datos, ya que `map()` y `apply()` ya brindan soluciones menos flexibles para la mayoría de los problemas. Aprenderá más en el Chapter 9.

5.3.3. Ejercicios

1. ¿Por qué este código tiene éxito sin errores ni advertencias?

```
x <- numeric()
out <- vector("list", length(x))
for (i in 1:length(x)) {
  out[i] <- x[i] ^ 2
}
out
```

2. Cuando se evalúa el siguiente código, ¿qué puede decir sobre el vector que se itera?

```
xs <- c(1, 2, 3)
for (x in xs) {
  xs <- c(xs, x * 2)
}
xs
#> [1] 1 2 3 2 4 6
```

3. ¿Qué le dice el siguiente código acerca de cuándo se actualiza el índice?

```
for (i in 1:3) {
  i <- i * 2
  print(i)
}
#> [1] 2
#> [1] 4
#> [1] 6
```

5.4. Respuestas de la prueba

- `if` trabaja con escalares; `ifelse()` trabaja con vectores.
- Cuando `x` es `TRUE`, `y` será `3`; cuando `FALSE`, `y` será `NULL`; cuando `NA`, la declaración if arrojará un error.
- Esta instrucción `switch()` hace uso de fallas, por lo que devolverá 2. Consulte los detalles en la Section 5.2.3.

6. Funciones

6.1. Introducción

Si está leyendo este libro, probablemente ya haya creado muchas funciones R y sepa cómo usarlas para reducir la duplicación en su código. En este capítulo, aprenderá cómo convertir ese conocimiento práctico e informal en una comprensión teórica más rigurosa. Y aunque verá algunos trucos y técnicas interesantes a lo largo del camino, tenga en cuenta que lo que aprenderá aquí será importante para comprender los temas más avanzados que se tratan más adelante en el libro.

Prueba

Responda las siguientes preguntas para ver si puede omitir este capítulo con seguridad. Puede encontrar las respuestas en la Section 6.9.

1. ¿Cuáles son los tres componentes de una función?
2. ¿Qué devuelve el siguiente código?

```
x <- 10
f1 <- function(x) {
  function() {
    x + 10
  }
}
f1(1)()
```

3. ¿Cómo escribirías normalmente este código?

```
`+`(1, `*`(2, 3))
```

4. ¿Cómo podría hacer que esta llamada sea más fácil de leer?

```
mean(, TRUE, x = c(1:10, NA))
```

5. ¿El siguiente código arroja un error cuando se ejecuta? ¿Por qué o por qué no?

```
f2 <- function(a, b) {
  a * 10
}
f2(10, stop("This is an error!"))
```

6. ¿Qué es una función infija? ¿Cómo lo escribes? ¿Qué es una función de reemplazo? ¿Cómo lo escribes?

7. ¿Cómo se asegura de que se produzca una acción de limpieza independientemente de cómo finalice una función?

Estructura

- La Section 6.2 describe los aspectos básicos de la creación de una función, los tres componentes principales de una función y la excepción a muchas reglas de funciones: funciones primitivas (que se implementan en C, no en R).
- La Section 6.3 analiza las fortalezas y debilidades de las tres formas de composición de funciones comúnmente utilizadas en el código R.
- La Section 6.4 le muestra cómo R encuentra el valor asociado con un nombre dado, es decir, las reglas del alcance léxico.

- La Section 6.5 está dedicada a una propiedad importante de los argumentos de función: solo se evalúan cuando se usan por primera vez.
- La Section 6.6 analiza el argumento especial `...`, que le permite pasar argumentos adicionales a otra función.
- La Section 6.7 analiza las dos formas principales en que una función puede salir y cómo definir un controlador de salida, código que se ejecuta al salir, independientemente de lo que lo active.
- La Section 6.8 le muestra las diversas formas en que R disfraza las llamadas a funciones ordinarias y cómo puede usar la forma de prefijo estándar para comprender mejor lo que está sucediendo.

6.2. Fundamentos de funciones

Para entender las funciones en R necesitas internalizar dos ideas importantes:

- Las funciones se pueden dividir en tres componentes: argumentos, cuerpo y entorno.

Hay excepciones a cada regla y, en este caso, hay una pequeña selección de funciones base "primitivas" que se implementan únicamente en C.

- Las funciones son objetos, al igual que los vectores son objetos.

6.2.1. Componentes de una función

Una función tiene tres partes.:

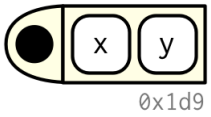
- Los formularios, `formals()`, la lista de argumentos que controlan cómo llamas a la función.

- El cuerpo, `body()`, el código dentro de la función.
- El entorno, `environment()`, la estructura de datos que determina cómo la función encuentra los valores asociados con los nombres.

Mientras que los formularios y el cuerpo se especifican explícitamente cuando crea una función, el entorno se especifica implícitamente, en función de *dónde* definió la función. El entorno de la función siempre existe, pero solo se imprime cuando la función no está definida en el entorno global.

```
f02 <- function(x, y) {
  # A comment
  x + y
}

formals(f02)
#> $x
#>
#>
#> $y

body(f02)
#> {
#>     x + y
#> }

environment(f02)
#> <environment: R_GlobalEnv>
```

Dibujaré funciones como en el siguiente diagrama. El punto negro de la izquierda es el entorno. Los dos bloques a la derecha son los argumentos de la función. No dibujaré el cuerpo, porque generalmente es grande y no te ayuda a entender la forma de la función.



Como todos los objetos en R, las funciones también pueden poseer cualquier cantidad de atributos, `attributes()`, adicionales. Un atributo utilizado por la base R es `srcref`, abreviatura de fuente de referencia. Apunta al código fuente utilizado para crear la función. El `srcref` se usa para imprimir porque, a diferencia de `body()`, contiene comentarios de código y otros formatos.

```
attr(f02, "srcref")
#> function(x, y) {
#>   # A comment
#>   x + y
#> }
```

6.2.2. Funciones primitivas

Hay una excepción a la regla de que una función tiene tres componentes. Las funciones primitivas, como `sum()` y `[`, llaman directamente al código C.

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")
`[`
#> .Primitive("[")
```

Ellas tienen el tipo `builtin` o tipo`special`.

```
typeof(sum)
#> [1] "builtin"
typeof(`[`)
#> [1] "special"
```

Estas funciones existen principalmente en C, no en R, por lo que sus `formals()`, `body()` y `environment()` son todos `NULL`:

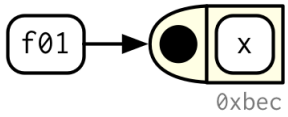
```
formals(sum)
#> NULL
body(sum)
#> NULL
environment(sum)
#> NULL
```

Las funciones primitivas solo se encuentran en el paquete base. Si bien tienen ciertas ventajas de rendimiento, este beneficio tiene un precio: son más difíciles de escribir. Por esta razón, R-core generalmente evita crearlos a menos que no haya otra opción.

6.2.3. Funciones de primera clase

Es muy importante comprender que las funciones de R son objetos por derecho propio, una propiedad del lenguaje a menudo denominada "funciones de primera clase". A diferencia de muchos otros lenguajes, no existe una sintaxis especial para definir y nombrar una función: simplemente crea un objeto de función (con `funtion`) y lo vincula a un nombre con `<-`:

```
f01 <- function(x) {
  sin(1 / x ^ 2)
}
```



Si bien casi siempre crea una función y luego la vincula a un nombre, el paso de vinculación no es obligatorio. Si elige no dar un nombre a una función, obtendrá una **función anónima**. Esto es útil cuando no vale la pena el esfuerzo de averiguar un nombre:

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

Una última opción es poner funciones en una lista:

```
funs <- list(
  half = function(x) x / 2,
  double = function(x) x * 2
)

funs$double(10)
#> [1] 20
```

En R, a menudo verá funciones llamadas cierres **closures**. Este nombre refleja el hecho de que las funciones de R capturan o encierran sus entornos, sobre los que aprenderá más en la Section 7.4.2.

6.2.4. Invocando una función

Normalmente llamas a una función colocando sus argumentos, entre paréntesis, después de su nombre: `mean(1:10, na.rm = TRUE)`. Pero, ¿qué sucede si ya tiene los argumentos en una estructura de datos?

```
args <- list(1:10, na.rm = TRUE)
```

En su lugar, puede usar `do.call()`: tiene dos argumentos. La función a llamar y una lista que contiene los argumentos de la función:

```
do.call(mean, args)
#> [1] 5.5
```

Volveremos a esta idea en la Section 19.6.

6.2.5. Ejercicios

1. Dado un nombre, como `"mean"`, `match.fun()` te permite encontrar una función. Dada una función, ¿puedes encontrar su nombre? ¿Por qué eso no tiene sentido en R?

2. Es posible (aunque normalmente no es útil) llamar a una función anónima. ¿Cuál de los dos enfoques siguientes es el correcto? ¿Por qué?

   ```
   function(x) 3()
   #> function(x) 3()
   (function(x) 3)()
   #> [1] 3
   ```

3. Una buena regla general es que una función anónima debería caber en una línea y no debería necesitar usar `{}`. Revisa tu código. ¿Dónde podría haber usado una función anónima en lugar de una función con nombre? ¿Dónde debería haber usado una función con nombre en lugar de una función anónima?

4. ¿Qué función te permite saber si un objeto es una función? ¿Qué función te permite saber si una función es una función primitiva?

5. Este código hace una lista de todas las funciones en el paquete base.

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

   Úsalo para responder las siguientes preguntas:

   a. ¿Qué función base tiene más argumentos?

   b. ¿Cuántas funciones base no tienen argumentos? ¿Qué tienen de especial esas funciones?

   c. ¿Cómo podrías adaptar el código para encontrar todas las funciones primitivas?

6. ¿Cuáles son los tres componentes importantes de una función?

7. ¿Cuándo la impresión de una función no muestra el entorno en el que se creó?

6.3. Composición de funciones

Base R proporciona dos formas de componer múltiples llamadas a funciones. Por ejemplo, imagina que quieres calcular la desviación estándar de la población usando `sqrt()` y `mean()` como bloques de construcción:

```
square <- function(x) x^2
deviation <- function(x) x - mean(x)
```

O anida las llamadas de función:

```
x <- runif(100)

sqrt(mean(square(deviation(x))))
#> [1] 0.274
```

O guarda los resultados intermedios como variables:

```
out <- deviation(x)
out <- square(out)
out <- mean(out)
out <- sqrt(out)
out
#> [1] 0.274
```

Tanto el paquete magrittr (Bache and Wickham 2014) y R base (a partir de la versión 4.1.0) proporcionan una tercera opción: el operador binario '%>%' y '|>' respectivamente, que se llama canalización y se pronuncia como "y luego".

```
library(magrittr)

x %>%
  deviation() %>%
  square() %>%
  mean() %>%
  sqrt()
#> [1] 0.274
```

O

```
x |>
  deviation() |>
  square() |>
  mean() |>
  sqrt()
#> [1] 0.274
```

`x |> f()` es equivalente a `f(x)`; `x |> f(y)` es equivalente a `f(x, y)`. La canalización le permite concentrarse en la composición de funciones de

alto nivel en lugar del flujo de datos de bajo nivel; el foco está en lo que se está haciendo (los verbos), más que en lo que se está modificando (los sustantivos). Este estilo es común en Haskell y F#, la principal inspiración para magrittr, y es el estilo predeterminado en lenguajes de programación basados en pilas como Forth y Factor. Para conocer más sobre las canalizaciones o pipe le recomiendo consultar la Sección 5.3 de *R para la Ciencia de Datos*

Cada una de las tres opciones tiene sus propias fortalezas y debilidades:

- El anidamiento, `f(g(x))`, es conciso y muy adecuado para secuencias cortas. Pero las secuencias más largas son difíciles de leer porque se leen al revés y de derecha a izquierda. Como resultado, los argumentos pueden extenderse a largas distancias creando el problema Dagwood sandwich.

- Objetos intermedios, `y <- f(x); g(y)`, requiere que nombres objetos intermedios. Esta es una fortaleza cuando los objetos son importantes, pero una debilidad cuando los valores son realmente intermedios.

- La canalización, `x |> f() |> g()`, le permite leer el código de manera directa de izquierda a derecha y no requiere que nombre objetos intermedios. Pero solo puede usarlo con secuencias lineales de transformaciones de un solo objeto. Asume que el lector entiende las canalizaciones.

La mayoría del código utilizará una combinación de los tres estilos. Las canalizaciones son más comunes en el código de análisis de datos, ya que gran parte de un análisis consiste en una secuencia de transformaciones de un objeto (como un marco de datos o un gráfico). Tiendo a usar canalizaciones con poca frecuencia en los paquetes; no porque sea una mala idea, sino porque a menudo es un ajuste menos natural.

6.4. Scooping léxico

En el Chapter 2, discutimos la asignación, el acto de vincular un nombre a un valor. Aquí hablaremos de **scoping**, el acto de encontrar el valor asociado con un nombre.

Las reglas básicas del scoping son bastante intuitivas y probablemente ya las haya internalizado, incluso si nunca las estudió explícitamente. Por ejemplo, ¿qué devolverá el siguiente código, 10 o 20?[1]

```
x <- 10
g01 <- function() {
  x <- 20
  x
}

g01()
```

En esta sección, aprenderá las reglas formales del alcance, así como algunos de sus detalles más sutiles. Una comprensión más profunda del alcance lo ayudará a usar herramientas de programación funcional más avanzadas y, eventualmente, incluso a escribir herramientas que traduzcan el código R a otros lenguajes.

R usa **scooping léxico**[2]: busca los valores de los nombres según cómo se define una función, no cómo se llama. "Léxico" o lexical, en inglés, es un término técnico de CS (**C**omputer **S**cience) que nos dice que las reglas de scooping utilizan un tiempo de análisis, en lugar de una estructura de tiempo de ejecución.

[1] Voy a "esconder" las respuestas a estos desafíos en las notas al pie. Intenta resolverlos antes de mirar la respuesta; esto le ayudará a recordar mejor la respuesta correcta. En este caso, `g01()` devolverá `20`.

[2] Las funciones que citan automáticamente uno o más argumentos pueden anular las reglas de alcance predeterminadas para implementar otras variedades de alcance. Aprenderá más sobre eso en el Chapter 20.

El scoopin léxico de R sigue 4 reglas principales:

- Enmascaramiento de nombres
- Funciones versus variables
- Un nuevo comienzo
- Búsqueda dinámica

6.4.1. Enmascaramiento de nombres

El principio básico del alcance léxico es que los nombres definidos dentro de una función enmascaran los nombres definidos fuera de una función. Esto se ilustra en el siguiente ejemplo.

```
x <- 10
y <- 20
g02 <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
g02()
#> [1] 1 2
```

Si un nombre no está definido dentro de una función, R busca un nivel superior.

```
x <- 2
g03 <- function() {
  y <- 1
  c(x, y)
}
g03()
#> [1] 2 1
```

```
# And this doesn't change the previous value of y
y
#> [1] 20
```

Las mismas reglas se aplican si una función se define dentro de otra función. Primero, R mira dentro de la función actual. Luego, busca dónde se definió esa función (y así sucesivamente, hasta llegar al entorno global). Finalmente, busca en otros paquetes cargados.

Ejecute el siguiente código en su cabeza, luego confirme el resultado ejecutando el código.[3]

```
x <- 1
g04 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
g04()
```

Las mismas reglas también se aplican a las funciones creadas por otras funciones, a las que llamo fabricas de funciones, el tema del Chapter 10.

6.4.2. Funciones versus variables

En R, las funciones son objetos ordinarios. Esto significa que las reglas de alcance descritas anteriormente también se aplican a las funciones:

[3] `g04()` resulta en `c(1, 2, 3)`.

```
g07 <- function(x) x + 1
g08 <- function() {
  g07 <- function(x) x + 100
  g07(10)
}
g08()
#> [1] 110
```

Sin embargo, cuando una función y una no función comparten el mismo nombre (por supuesto, deben residir en diferentes entornos), la aplicación de estas reglas se vuelve un poco más complicada. Cuando usa un nombre en una llamada de función, R ignora los objetos que no son funciones cuando busca ese valor. Por ejemplo, en el siguiente código, `g09` toma dos valores diferentes:

```
g09 <- function(x) x + 100
g10 <- function() {
  g09 <- 10
  g09(g09)
}
g10()
#> [1] 110
```

Para que conste, usar el mismo nombre para diferentes cosas es confuso y ¡es mejor evitarlo!

6.4.3. Un nuevo comienzo

¿Qué sucede con los valores entre invocaciones de una función? Considere el siguiente ejemplo. ¿Qué sucederá la primera vez que ejecute esta función? ¿Qué pasará la segunda vez?[4] (Si no ha visto `exists()` antes,

[4] `g11()` devuelve `1` cada vez que se llama.

devuelve `TRUE` si hay una variable con ese nombre y devuelve `FALSE` si no.)

```
g11 <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}

g11()
g11()
```

Puede que te sorprenda que `g11()` siempre devuelve el mismo valor. Esto sucede porque cada vez que se llama a una función, se crea un nuevo entorno para albergar su ejecución. Esto significa que una función no tiene forma de saber qué sucedió la última vez que se ejecutó; cada invocación es completamente independiente. Veremos algunas formas de evitar esto en la Section 10.2.4.

6.4.4. Búsqueda dinámica

El scooping léxico determina dónde, pero no cuándo buscar valores. R busca valores cuando se ejecuta la función, no cuando se crea la función. Juntas, estas dos propiedades nos dicen que la salida de una función puede diferir dependiendo de los objetos fuera del entorno de la función:

```
g12 <- function() x + 1
x <- 15
g12()
```

```
#> [1] 16

x <- 20
g12()
#> [1] 21
```

Este comportamiento puede ser bastante molesto. Si comete un error ortográfico en su código, no recibirá un mensaje de error cuando cree la función. Y dependiendo de las variables definidas en el entorno global, es posible que ni siquiera reciba un mensaje de error cuando ejecute la función.

Para detectar este problema, utilice `codetools::findGlobals()`. Esta función enumera todas las dependencias externas (símbolos independientes) dentro de una función:

```
codetools::findGlobals(g12)
#> [1] "+" "x"
```

Para resolver este problema, puede cambiar manualmente el entorno de la función a `emptyenv()`, un entorno que no contiene nada:

```
environment(g12) <- emptyenv()
g12()
#> Error in x + 1: could not find function "+"
```

El problema y su solución revelan por qué existe este comportamiento aparentemente indeseable: R se basa en el scooping léxico para encontrar *todo*, desde lo obvio, como `mean()`, hasta lo menos obvio, como `+` o incluso `{`. Esto le da a las reglas de scooping de R una hermosa simplicidad.

6.4.5. Ejercicios

1. ¿Qué devuelve el siguiente código? ¿Por qué? Describa cómo se interpreta cada una de las tres `c`.

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

2. ¿Cuáles son los cuatro principios que rigen cómo R busca valores?

3. ¿Qué devuelve la siguiente función? Haga una predicción antes de ejecutar el código usted mismo.

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

6.5. Evaluación perezosa

En R, los argumentos de función se **evalúan perezosamente**: solo se evalúan si se accede a ellos. Por ejemplo, este código no genera un error porque `x` nunca se usa:

```
h01 <- function(x) {
  10
}
```

```
h01(stop("This is an error!"))
#> [1] 10
```

Esta es una característica importante porque le permite hacer cosas como incluir cálculos potencialmente costosos en los argumentos de la función que solo se evaluarán si es necesario.

6.5.1. Promesas

La evaluación perezosa está impulsada por una estructura de datos llamada **promesa** o (menos comúnmente) un thunk. Es una de las características que hace de R un lenguaje de programación tan interesante (volveremos a las promesas en la Section 20.3).

Una promesa tiene tres componentes:

- Una expresión, como `x + y`, que da lugar al cálculo retrasado.
- Un entorno donde se debe evaluar la expresión, es decir, el entorno donde se llama a la función. Esto asegura que la siguiente función devuelva 11, no 101:

  ```
  y <- 10
  h02 <- function(x) {
    y <- 100
    x + 1
  }

  h02(y)
  #> [1] 11
  ```

  Esto también significa que cuando realiza una asignación dentro de una llamada a una función, la variable se vincula fuera de la función, no dentro de ella.

```
h02(y <- 1000)
#> [1] 1001
y
#> [1] 1000
```

- Un valor, que se calcula y almacena en caché la primera vez que se accede a una promesa cuando la expresión se evalúa en el entorno especificado. Esto asegura que la promesa se evalúe como máximo una vez, y es por eso que solo ve "Calculando..." impreso una vez en el siguiente ejemplo.

```
double <- function(x) {
  message("Calculating...")
  x * 2
}

h03 <- function(x) {
  c(x, x)
}

h03(double(20))
#> Calculating...
#> [1] 40 40
```

No puede manipular promesas con código R. Las promesas son como un estado cuántico: cualquier intento de inspeccionarlas con código R forzará una evaluación inmediata, haciendo que la promesa desaparezca. Más adelante, en la Section 20.3, aprenderá sobre las quosures, que convierten las promesas en un objeto R donde puede inspeccionar fácilmente la expresión y el entorno.

6.5.2. Argumentos por defecto

Gracias a la evaluación perezosa, los valores predeterminados se pueden definir en términos de otros argumentos, o incluso en términos de variables definidas más adelante en la función:

```
h04 <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100

  c(x, y, z)
}

h04()
#> [1]   1   2 110
```

Muchas funciones base de R usan esta técnica, pero no la recomiendo. Hace que el código sea más difícil de entender: para predecir *qué* se devolverá, necesita saber el orden exacto en el que se evalúan los argumentos predeterminados.

El entorno de evaluación es ligeramente diferente para los argumentos predeterminados y proporcionados por el usuario, ya que los argumentos predeterminados se evalúan dentro de la función. Esto significa que llamadas aparentemente idénticas pueden producir resultados diferentes. Es más fácil ver esto con un ejemplo extremo:

```
h05 <- function(x = ls()) {
  a <- 1
  x
}

# ls() evaluated inside h05:
h05()
```

```
#> [1] "a" "x"

# ls() evaluated in global environment:
h05(ls())
#> [1] "h05"
```

6.5.3. Argumentos faltantes

Para determinar si el valor de un argumento proviene del usuario o de un valor predeterminado, puede usar `missing()`:

```
h06 <- function(x = 10) {
  list(missing(x), x)
}
str(h06())
#> List of 2
#>  $ : logi TRUE
#>  $ : num 10
str(h06(10))
#> List of 2
#>  $ : logi FALSE
#>  $ : num 10
```

`missing()` sin embargo, es mejor usarlo con moderación. Tome `sample()`, como ejemplo. ¿Cuántos argumentos se requieren?

```
args(sample)
#> function (x, size, replace = FALSE, prob = NULL)
#> NULL
```

Parece que tanto `x` como `size` son necesarios, pero si no se proporciona `size`, `sample()` usa `missing()` para proporcionar un valor predetermi-

nado. Si tuviera que volver a escribir la muestra, usaría un `NULL` explícito para indicar que no se requiere `size` pero se puede proporcionar:

```
sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
  if (is.null(size)) {
    size <- length(x)
  }

  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

Con el patrón binario creado por la función infija `%||%`, que usa el lado izquierdo si no es `NULL` y el lado derecho en caso contrario, podemos simplificar aún más `sample()`:

```
`%||%` <- function(lhs, rhs) {
  if (!is.null(lhs)) {
    lhs
  } else {
    rhs
  }
}

sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
  size <- size %||% length(x)
  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

Debido a la evaluación perezosa, no necesita preocuparse por cálculos innecesarios: el lado derecho de `%||%` solo se evaluará si el lado izquierdo es `NULL`.

6.5.4. Ejercicios

1. ¿Qué propiedad importante de `&&` hace que `x_ok()` funcione?

```
x_ok <- function(x) {
  !is.null(x) && length(x) == 1 && x > 0
}

x_ok(NULL)
#> [1] FALSE
x_ok(1)
#> [1] TRUE
x_ok(1:3)
#> [1] FALSE
```

   ¿Qué es diferente con este código? ¿Por qué este comportamiento es indeseable aquí?

```
x_ok <- function(x) {
  !is.null(x) & length(x) == 1 & x > 0
}

x_ok(NULL)
#> logical(0)
x_ok(1)
#> [1] TRUE
x_ok(1:3)
#> [1] FALSE FALSE FALSE
```

2. ¿Qué devuelve esta función? ¿Por qué? ¿Qué principio ilustra?

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

3. ¿Qué devuelve esta función? ¿Por qué? ¿Qué principio ilustra?

```
y <- 10
f1 <- function(x = {y <- 1; 2}, y = 0) {
  c(x, y)
}
f1()
y
```

4. En `hist()`, el valor predeterminado de `xlim` es `range(breaks)`, el valor predeterminado de `breaks` es `"Sturges"`, y

```
range("Sturges")
#> [1] "Sturges" "Sturges"
```

   Explique cómo funciona `hist()` para obtener un valor `xlim` correcto.

5. Explique por qué funciona esta función. ¿Por qué es confuso?

```
show_time <- function(x = stop("Error!")) {
  stop <- function(...) Sys.time()
  print(x)
}
show_time()
#> [1] "2024-08-17 13:27:30 UTC"
```

6. ¿Cuántos argumentos se requieren al llamar a `library()`?

6.6. ... (punto-punto-punto)

Las funciones pueden tener un argumento especial `...` (pronunciado punto-punto-punto). Con él, una función puede tomar cualquier número de argumentos adicionales. En otros lenguajes de programación, este tipo de argumento a menudo se llama *varargs* (abreviatura de argumentos variables), y una función que lo usa se dice que es variable.

También puede usar `...` para pasar esos argumentos adicionales a otra función.

```
i01 <- function(y, z) {
  list(y = y, z = z)
}

i02 <- function(x, ...) {
  i01(...)
}

str(i02(x = 1, y = 2, z = 3))
#> List of 2
#>  $ y: num 2
#>  $ z: num 3
```

Usando una forma especial, `..N`, es posible (pero rara vez útil) referirse a elementos de `...` por posición:

```
i03 <- function(...) {
  list(first = ..1, third = ..3)
}
str(i03(1, 2, 3))
#> List of 2
#>  $ first: num 1
#>  $ third: num 3
```

Más útil es `list(...)`, que evalúa los argumentos y los almacena en una lista:

```
i04 <- function(...) {
  list(...)
}
```

```
str(i04(a = 1, b = 2))
#> List of 2
#>  $ a: num 1
#>  $ b: num 2
```

(Consulte también `rlang::list2()` para admitir el empalme e ignorar silenciosamente las comas finales, y `rlang::enquos()` para capturar argumentos no evaluados, el tema de [cuasicotización].)

Hay dos usos principales de `...`, a los cuales volveremos más adelante en el libro:

- Si su función toma una función como argumento, querrá alguna forma de pasar argumentos adicionales a esa función. En este ejemplo, `lapply()` usa `...` para pasar `na.rm` a `mean()`:

  ```
  x <- list(c(1, 3, NA), c(4, NA, 6))
  str(lapply(x, mean, na.rm = TRUE))
  #> List of 2
  #>  $ : num 2
  #>  $ : num 5
  ```

  Volveremos a esta técnica en la Section 9.2.3.

- Si su función es genérica de S3, necesita alguna forma de permitir que los métodos tomen argumentos adicionales arbitrarios. Por ejemplo, tome la función `print()`. Debido a que existen diferentes opciones para imprimir según el tipo de objeto, no hay forma de especificar previamente todos los argumentos posibles y `...` permite que los métodos individuales tengan diferentes argumentos:

  ```
  print(factor(letters), max.levels = 4)

  print(y ~ x, showEnv = TRUE)
  ```

  Volveremos a este uso de `...` en la Section 13.4.3.

Usar ... tiene dos desventajas:

- Cuando lo usa para pasar argumentos a otra función, debe explicar cuidadosamente al usuario dónde van esos argumentos. Esto hace que sea difícil entender lo que puedes hacer con funciones como `lapply()` y `plot()`.

- Un argumento mal escrito no generará un error. Esto facilita que los errores tipográficos pasen desapercibidos:

```
sum(1, 2, NA, na_rm = TRUE)
#> [1] NA
```

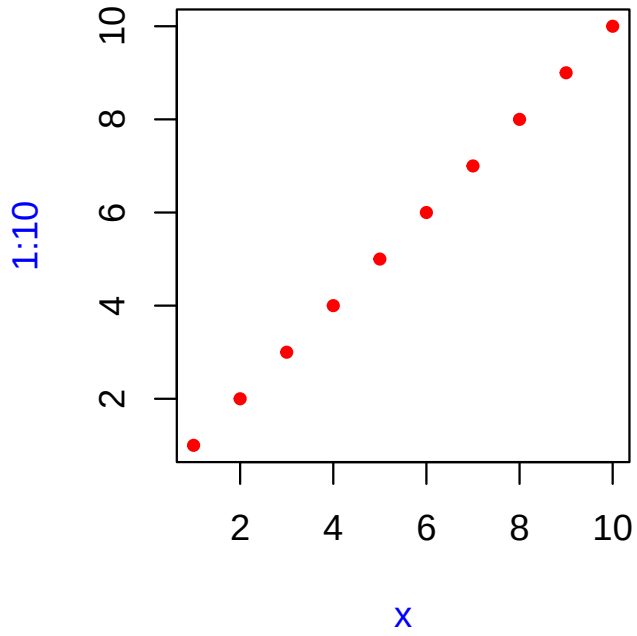
6.6.1. Ejercicios

1. Explique los siguientes resultados:

```
sum(1, 2, 3)
#> [1] 6
mean(1, 2, 3)
#> [1] 1

sum(1, 2, 3, na.omit = TRUE)
#> [1] 7
mean(1, 2, 3, na.omit = TRUE)
#> [1] 1
```

2. Explique cómo encontrar la documentación para los argumentos con nombre en la siguiente llamada de función:

```
plot(1:10, col = "red", pch = 20, xlab = "x", col.lab = "blue")
```



3. ¿Por qué `plot(1:10, col = "red")` solo colorea los puntos, no los ejes ni las etiquetas? Lea el código fuente de `plot.default()` para averiguarlo.

6.7. Salir de una función

La mayoría de las funciones salen de una de dos maneras[5]: o devuelven un valor, lo que indica el éxito, o arrojan un error, lo que indica el fracaso. Esta sección describe los valores devueltos (implícitos frente a explícitos; visibles frente a invisibles), analiza brevemente los errores y presenta los

[5] Las funciones pueden salir de otras formas más esotéricas, como señalar una condición detectada por un controlador de salida, invocar un reinicio o presionar "Q" en un navegador interactivo.

controladores de salida, que le permiten ejecutar código cuando una función sale.

6.7.1. Rendimientos implícitos versus explícitos

Hay dos formas en que una función puede devolver un valor:

- Implícitamente, donde la última expresión evaluada es el valor de retorno:

```
j01 <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
j01(5)
#> [1] 0
j01(15)
#> [1] 10
```

- Explícitamente, llamando `return()`:

```
j02 <- function(x) {
  if (x < 10) {
    return(0)
  } else {
    return(10)
  }
}
```

6.7.2. Valores invisibles

La mayoría de las funciones regresan visiblemente: llamar a la función en un contexto interactivo imprime el resultado.

```
j03 <- function() 1
j03()
#> [1] 1
```

Sin embargo, puede evitar la impresión automática aplicando `invisible()` al último valor:

```
j04 <- function() invisible(1)
j04()
```

Para verificar que este valor existe, puede imprimirlo explícitamente o envolverlo entre paréntesis:

```
print(j04())
#> [1] 1

(j04())
#> [1] 1
```

Alternativamente, puedes usar `withVisible()` para devolver el valor y un indicador de visibilidad:

```
str(withVisible(j04()))
#> List of 2
#>  $ value  : num 1
#>  $ visible: logi FALSE
```

La función más común que regresa invisiblemente es `<-`:

```
a <- 2
(a <- 2)
#> [1] 2
```

Esto es lo que hace posible encadenar asignaciones:

```
a <- b <- c <- d <- 2
```

En general, cualquier función llamada principalmente por un efecto secundario (como `<-`, `print()` o `plot()`) debería devolver un valor invisible (normalmente el valor del primer argumento).

6.7.3. Errores

Si una función no puede completar su tarea asignada, debería arrojar un error con `stop()`, que finaliza inmediatamente la ejecución de la función.

```
j05 <- function() {
  stop("I'm an error")
  return(10)
}
j05()
#> Error in j05(): I'm an error
```

Un error indica que algo salió mal y obliga al usuario a solucionar el problema. Algunos lenguajes (como C, Go y Rust) se basan en valores de retorno especiales para indicar problemas, pero en R siempre debe arrojar un error. Aprenderá más sobre los errores y cómo manejarlos en el Chapter 8.

6.7.4. Controladores de salida

A veces, una función necesita realizar cambios temporales en el estado global. Pero tener que limpiar esos cambios puede ser doloroso (¿qué sucede si hay un error?). Para asegurarse de que estos cambios se deshagan y que el estado global se restablezca sin importar cómo salga una función, use `on.exit()` para configurar un **controlador de salida**. El siguiente ejemplo simple muestra que el controlador de salida se ejecuta independientemente de si la función sale normalmente o con un error.

```
j06 <- function(x) {
  cat("Hello\n")
  on.exit(cat("Goodbye!\n"), add = TRUE)

  if (x) {
    return(10)
  } else {
    stop("Error")
  }
}

j06(TRUE)
#> Hello
#> Goodbye!
#> [1] 10

j06(FALSE)
#> Hello
#> Error in j06(FALSE): Error
#> Goodbye!
```

Establezca siempre `add = TRUE` cuando use `on.exit()`. Si no lo hace, cada llamada a `on.exit()` sobrescribirá el controlador de salida anterior.

Incluso cuando solo se registra un único controlador, es una buena práctica configurar `add = TRUE` para que no se lleve sorpresas desagradables si luego agrega más controladores de salida.

`on.exit()` es útil porque le permite colocar el código de limpieza directamente al lado del código que requiere limpieza:

```
cleanup <- function(dir, code) {
  old_dir <- setwd(dir)
  on.exit(setwd(old_dir), add = TRUE)

  old_opt <- options(stringsAsFactors = FALSE)
  on.exit(options(old_opt), add = TRUE)
}
```

Junto con la evaluación perezosa, esto crea un patrón muy útil para ejecutar un bloque de código en un entorno alterado:

```
with_dir <- function(dir, code) {
  old <- setwd(dir)
  on.exit(setwd(old), add = TRUE)

  force(code)
}

getwd()
#> [1] "/home/runner/work/adv-res/adv-res"
with_dir("~", getwd())
#> [1] "/home/runner"
```

El uso de `force()` no es estrictamente necesario aquí ya que simplemente referirse al `código` forzará su evaluación. Sin embargo, usar `force()` deja muy claro que estamos forzando deliberadamente la ejecución. Aprenderá otros usos de `force()` en el Chapter 10.

El paquete withr (Hester et al. 2018) proporciona una colección de otras funciones para configurar un estado temporal.

En R 3.4 y versiones anteriores, las expresiones `on.exit()` siempre se ejecutan en orden de creación:

```
j08 <- function() {
  on.exit(message("a"), add = TRUE)
  on.exit(message("b"), add = TRUE)
}
j08()
#> a
#> b
```

Esto puede dificultar un poco la limpieza si es necesario realizar algunas acciones en un orden específico; por lo general, desea que se ejecute primero la expresión añadida más reciente. En R 3.5 y versiones posteriores, puede controlar esto configurando `after = FALSE`:

```
j09 <- function() {
  on.exit(message("a"), add = TRUE, after = FALSE)
  on.exit(message("b"), add = TRUE, after = FALSE)
}
j09()
#> b
#> a
```

6.7.5. Ejercicios

1. ¿Qué devuelve `load()`? ¿Por qué normalmente no ves estos valores?

2. ¿Qué devuelve `write.table()`? ¿Qué sería más útil?

3. ¿Cómo se compara el parámetro `chdir` de `source()` con `with_dir()`? ¿Por qué preferirías uno a otro?

4. Escriba una función que abra un dispositivo de gráficos, ejecute el código proporcionado y cierre el dispositivo de gráficos (siempre, independientemente de si el código de trazado funciona o no).

5. Podemos usar `on.exit()` para implementar una versión simple de `capture.output()`.

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

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

  force(code)
  readLines(temp)
}
capture.output2(cat("a", "b", "c", sep = "\n"))
#> [1] "a" "b" "c"
```

Compara `capture.output()` con `capture.output2()`. ¿Cómo difieren las funciones? ¿Qué características he eliminado para que las ideas clave sean más fáciles de ver? ¿Cómo he reescrito las ideas clave para que sean más fáciles de entender?

6.8. Formas de función

Para comprender los cálculos en R, dos lemas son útiles:

- Todo lo que existe es un objeto.
- Todo lo que sucede es una llamada de función.

— John Chambers

Si bien todo lo que sucede en R es el resultado de una llamada de función, no todas las llamadas tienen el mismo aspecto. Las llamadas a funciones vienen en cuatro variedades:

- **prefija**: el nombre de la función viene antes de sus argumentos, como `foofy(a, b, c)`. Estos constituyen la mayoría de las llamadas a funciones en R.
- **infija**: el nombre de la función viene entre sus argumentos, como `x + y`. Las formas infijas se utilizan para muchos operadores matemáticos y para funciones definidas por el usuario que comienzan y terminan con `%`.
- **reemplazo**: funciones que reemplazan valores por asignación, como `names(df) <- c("a", "b", "c")`. En realidad, parecen funciones de prefijo.
- **especial**: funciones como `[[`, `if` y `for`. Si bien no tienen una estructura consistente, juegan papeles importantes en la sintaxis de R.

Si bien hay cuatro formas, en realidad solo necesita una porque cualquier llamada se puede escribir en forma de prefijo. Demostraré esta propiedad y luego aprenderá sobre cada una de las formas.

6.8.1. Reescritura en forma de prefijo

Una propiedad interesante de R es que cada infijo, reemplazo o forma especial se puede reescribir en forma de prefijo. Hacerlo es útil porque lo ayuda a comprender mejor la estructura del lenguaje, le brinda el nombre real de cada función y le permite modificar esas funciones para divertirse y obtener ganancias.

El siguiente ejemplo muestra tres pares de llamadas equivalentes, reescribiendo una forma de infijo, una forma de reemplazo y una forma especial en forma de prefijo.

```
x + y
`+`(x, y)

names(df) <- c("x", "y", "z")
`names<-`(df, c("x", "y", "z"))

for(i in 1:10) print(i)
`for`(i, 1:10, print(i))
```

¡Sorprendentemente, en R, `for` se puede llamar como una función normal! Lo mismo es cierto básicamente para todas las operaciones en R, lo que significa que conocer el nombre de la función de una función sin prefijo le permite anular su comportamiento. Por ejemplo, si alguna vez te sientes particularmente mal, ejecuta el siguiente código mientras un amigo no está en su computadora. Introducirá un error divertido: el 10% de las veces, agregará 1 a cualquier cálculo numérico dentro de los paréntesis.

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
#>  [1] 3 3 3 3 3 3 3 3 3 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
#> [33] 3 3 3 3 3 3 4 3 4 3 3 3 3 4 3 3 3 3
rm("(")
```

Por supuesto, anular funciones integradas como esta es una mala idea, pero, como aprenderá en la Section 21.2.5, es posible aplicarlo solo a bloques de código seleccionados. Esto proporciona un enfoque limpio y elegante para escribir idiomas específicos de dominio y traductores a otros idiomas.

Una aplicación más útil surge cuando se utilizan herramientas de programación funcional. Por ejemplo, podría usar `lapply()` para agregar 3 a cada elemento de una lista definiendo primero una función `add()`:

```
add <- function(x, y) x + y
lapply(list(1:3, 4:5), add, 3)
#> [[1]]
#> [1] 4 5 6
#>
#> [[2]]
#> [1] 7 8
```

Pero también podemos obtener el mismo resultado simplemente confiando en la función `+` existente:

```
lapply(list(1:3, 4:5), `+`, 3)
#> [[1]]
#> [1] 4 5 6
#>
#> [[2]]
#> [1] 7 8
```

Exploraremos esta idea en detalle en la Chapter 9.

6.8.2. Forma de prefijo

La forma de prefijo es la forma más común en el código R y, de hecho, en la mayoría de los lenguajes de programación. Las llamadas de prefijo

en R son un poco especiales porque puede especificar argumentos de tres maneras:

- Por posición, como `help(mean)`.
- Usando coincidencias parciales, como `help(top = mean)`.
- Por nombre, como `help(topic = mean)`.

Como se ilustra en el siguiente fragmento, los argumentos se comparan por nombre exacto, luego con prefijos únicos y finalmente por posición.

```
k01 <- function(abcdef, bcde1, bcde2) {
  list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
str(k01(1, 2, 3))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3
str(k01(2, 3, abcdef = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# Puede abreviar nombres de argumentos largos:
str(k01(2, 3, a = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# Pero esto no funciona porque la abreviatura es ambigua.
str(k01(1, 3, b = 1))
#> Error in k01(1, 3, b = 1): argument 3 matches multiple formal arguments
```

En general, use la coincidencia posicional solo para los primeros uno o dos argumentos; serán los más utilizados y la mayoría de los lectores sabrán cuáles son. Evite el uso de coincidencias posicionales para argumentos que se usan con menos frecuencia y nunca use coincidencias parciales. Desafortunadamente, no puede deshabilitar la coincidencia parcial, pero puede convertirla en una advertencia con la opción `warnPartialMatchArgs`:

```
options(warnPartialMatchArgs = TRUE)
x <- k01(a = 1, 2, 3)
#> Warning in k01(a = 1, 2, 3): partial argument match of 'a' to
#> 'abcdef'
```

6.8.3. Funciones infijas

Las funciones infijas obtienen su nombre del hecho de que el nombre de la función se encuentra entre sus argumentos y, por lo tanto, tienen dos argumentos. R viene con una serie de operadores infijos incorporados: `:`, `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&`, `&&`, `|`, `||`, `~`, `<-`, y `<<-`. También puede crear sus propias funciones infijas que comiencen y terminen con `%`. Base R usa este patrón para definir `%%`, `%*%`, `%/%`, `%in%`, `%o%`, y `%x%`.

Definir su propia función de infijo es simple. Creas una función de dos argumentos y la vinculas a un nombre que comienza y termina con `%`:

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

Los nombres de las funciones infijas son más flexibles que las funciones regulares de R: pueden contener cualquier secuencia de caracteres excepto `%`. Deberá escapar cualquier carácter especial en la cadena utilizada para definir la función, pero no cuando la llame:

```
`% %` <- function(a, b) paste(a, b)
`%/\\%` <- function(a, b) paste(a, b)

"a" % % "b"
#> [1] "a b"
"a" %/\% "b"
#> [1] "a b"
```

Las reglas de precedencia predeterminadas de R significan que los operadores infijos se componen de izquierda a derecha:

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

Hay dos funciones infijas especiales que se pueden llamar con un solo argumento: `+` y `-`.

```
-1
#> [1] -1
+10
#> [1] 10
```

6.8.4. Funciones de reemplazo

Las funciones de reemplazo actúan como si modificaran sus argumentos en su lugar y tienen el nombre especial `xxx<-`. Deben tener argumentos llamados `x` y `value`, y deben devolver el objeto modificado. Por ejemplo, la siguiente función modifica el segundo elemento de un vector:

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

Las funciones de reemplazo se utilizan colocando la llamada de función en el lado izquierdo de `<-`:

```
x <- 1:10
second(x) <- 5L
x
#>  [1]  1  5  3  4  5  6  7  8  9 10
```

Digo que actúan como si modificaran sus argumentos en el lugar porque, como se explica en la Section 2.5, en realidad crean una copia modificada. Podemos ver eso usando `tracemem()`:

```
x <- 1:10
tracemem(x)
#> <0x7ffae71bd880>

second(x) <- 6L
#> tracemem[0x7ffae71bd880 -> 0x7ffae61b5480]:
#> tracemem[0x7ffae61b5480 -> 0x7ffae73f0408]: second<-
```

Si su función de reemplazo necesita argumentos adicionales, colóquelos entre `x` y `value`, y llame a la función de reemplazo con argumentos adicionales a la izquierda:

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
```

```
}
modify(x, 1) <- 10
x
#>  [1] 10  5  3  4  5  6  7  8  9 10
```

Cuando escribe `modify(x, 1) <- 10`, detrás de escena R lo convierte en:

```
x <- `modify<-`(x, 1, 10)
```

La combinación de reemplazo con otras funciones requiere una traducción más compleja. Por ejemplo:

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

names(x)[2] <- "two"
names(x)
#> [1] "a"   "two" "c"
```

se traduce en:

```
`*tmp*` <- x
x <- `names<-`(`*tmp*`, `[<-`(names(`*tmp*`), 2, "two"))
rm(`*tmp*`)
```

(Sí, realmente crea una variable local llamada `*tmp*`, que se elimina después.)

6.8.5. Formas especiales

Finalmente, hay un montón de características del lenguaje que normalmente se escriben de formas especiales, pero que también tienen formas de prefijo. Estos incluyen paréntesis:

- `(x)` (`` `(`(x) ``)
- `{x}` (`` `{`(x) ``).

Los operadores de subconjuntos:

- `x[i]` (`` `[`(x, i) ``)
- `x[[i]]` (`` `[[`(x, i) ``)

Y las herramientas de control de flujo:

- `if (cond) true` (`` `if`(cond, true) ``)
- `if (cond) true else false` (`` `if`(cond, true, false) ``)
- `for(var in seq) action` (`` `for`(var, seq, action) ``)
- `while(cond) action` (`` `while`(cond, action) ``)
- `repeat expr` (`` `repeat`(expr) ``)
- `next` (`` `next`() ``)
- `break` (`` `break`() ``)

Finalmente, la más compleja es la función `function`:

- `function(arg1, arg2) {body}` (`` `function`(alist(arg1, arg2), body, env) ``)

Conocer el nombre de la función que subyace en una forma especial es útil para obtener documentación: `?(` es un error de sintaxis; `` ?`(` `` le dará la documentación para los paréntesis.

Todas las formas especiales se implementan como funciones primitivas (es decir, en C); esto significa que imprimir estas funciones no es informativo:

```
`for`
#> .Primitive("for")
```

6.8.6. Ejercicios

1. Reescriba los siguientes fragmentos de código en forma de prefijo:

```
1 + 2 + 3

1 + (2 + 3)

if (length(x) <= 5) x[[5]] else x[[n]]
```

2. Aclare la siguiente lista de llamadas a funciones impares:

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

3. Explique por qué falla el siguiente código:

```
modify(get("x"), 1) <- 10
#> Error: target of assignment expands to non-language object
```

4. Cree una función de reemplazo que modifique una ubicación aleatoria en un vector.

5. Escriba su propia versión de + que pegue sus entradas juntas si son vectores de caracteres, pero se comporte como de costumbre en caso contrario. En otras palabras, haz que este código funcione:

```
1 + 2
#> [1] 3

"a" + "b"
#> [1] "ab"
```

6. Cree una lista de todas las funciones de reemplazo que se encuentran en el paquete base. ¿Cuáles son funciones primitivas? (Sugerencia: utilice `apropos()`.)
7. ¿Cuáles son los nombres válidos para las funciones infijas creadas por el usuario?
8. Cree un operador infijo `xor()`.
9. Cree versiones infijas de las funciones de conjunto `intersect()`, `union()` y `setdiff()`. Puede llamarlos `%n%`, `%u%` y `%/%` para que coincidan con las convenciones de las matemáticas.

6.9. Respuestas de la Prueba

1. Los tres componentes de una función son su cuerpo, argumentos y entorno.
2. `f1(1)()` devuelve 11.
3. Normalmente lo escribirías en estilo infijo: `1 + (2 * 3)`.
4. Reescribiendo la llamada a `mean(c(1:10, NA), na.rm = TRUE)` es más fácil de entender.
5. No, no arroja un error porque el segundo argumento nunca se usa, por lo que nunca se evalúa.
6. Vea Secciones Section 6.8.3 y Section 6.8.4.
7. Usas `on.exit()`; vea la Section 6.7.4 para más detalles.

7. Entornos

7.1. Introduction

El entorno es la estructura de datos que impulsa el alcance. Este capítulo profundiza en los entornos, describe su estructura en profundidad y los usa para mejorar su comprensión de las cuatro reglas de scoping descritas en la Section 6.4. Comprender los entornos no es necesario para el uso diario de R. Pero es importante comprenderlos porque impulsan muchas funciones importantes de R, como el scoping léxico, los espacios de nombres y las clases R6, e interactúan con la evaluación para brindarle herramientas poderosas para crear dominios. lenguajes específicos, como dplyr y ggplot2.

Prueba

Si puede responder correctamente las siguientes preguntas, ya conoce los temas más importantes de este capítulo. Puede encontrar las respuestas al final del capítulo en la Section 7.7.

1. Enumere al menos tres formas en que un entorno difiere de una lista.
2. ¿Cuál es el padre del medio ambiente global? ¿Cuál es el único entorno que no tiene un padre?
3. ¿Qué es el entorno envolvente de una función? ¿Por qué es importante?

4. ¿Cómo determina el entorno desde el que se llamó a una función?
5. ¿En qué se diferencian `<-` y `<<-`?

Estructura

- La Section 7.2 le presenta las propiedades básicas de un entorno y le muestra cómo crear el suyo propio.
- La Section 7.3 proporciona una plantilla de funciones para computar con entornos, ilustrando la idea con una función útil.
- La Section 7.4 describe entornos utilizados para fines especiales: para paquetes, dentro de funciones, para espacios de nombres y para la ejecución de funciones.
- La Section 7.5 explica el último entorno importante: el entorno de la persona que llama. Esto requiere que aprenda sobre la pila de llamadas, que describe cómo se llamó a una función. Habrás visto la pila de llamadas si alguna vez llamaste a `traceback()` para ayudar en la depuración.
- La Section 7.6 analiza brevemente tres lugares donde los entornos son estructuras de datos útiles para resolver otros problemas.

Requisitos previos

Este capítulo utilizará las funciones rlang para trabajar con entornos, ya que nos permite centrarnos en la esencia de los entornos, en lugar de los detalles secundarios.

```
library(rlang)
```

Las funciones `env_` en rlang están diseñadas para trabajar con la canalización: todas toman un entorno como primer argumento, y muchas también devuelven un entorno. No usaré la canalización en este capítulo con el fin de mantener el código lo más simple posible, pero debería considerarlo para su propio código.

7.2. Conceptos básicos de entornos

En general, un entorno es similar a una lista con nombre, con cuatro excepciones importantes:

- Cada nombre debe ser único.
- Los nombres de un entorno no están ordenados.
- Un entorno tiene un padre.
- Los entornos no se copian cuando se modifican.

Exploremos estas ideas con código e imágenes.

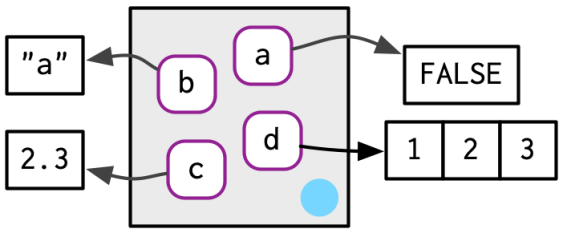
7.2.1. Lo esencial

Para crear un entorno, utilice `rlang::env()`. Funciona como `list()`, tomando un conjunto de pares nombre-valor:

```
e1 <- env(
  a = FALSE,
  b = "a",
  c = 2.3,
  d = 1:3,
)
```

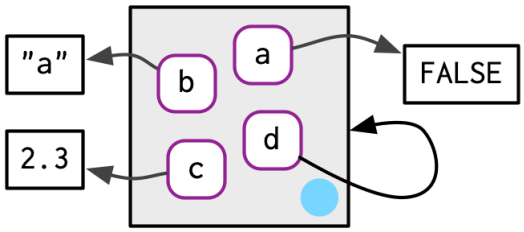
Utilice `new.env()` para crear un nuevo entorno. Ignora los parámetros `hash` y `size`; no son necesarios. No puede crear y definir valores simultáneamente; use `$<-`, como se muestra a continuación.

El trabajo de un entorno es asociar, o **vincular**, un conjunto de nombres a un conjunto de valores. Puede pensar en un entorno como una bolsa de nombres, sin orden implícito (es decir, no tiene sentido preguntar cuál es el primer elemento en un entorno). Por esa razón, dibujaremos el entorno así:



Como se discutió en la Section 2.5.2, los entornos tienen una semántica de referencia: a diferencia de la mayoría de los objetos R, cuando los modifica, los modifica en su lugar y no crea una copia. Una implicación importante es que los entornos pueden contenerse a sí mismos.

```
e1$d <- e1
```



Imprimir un entorno solo muestra su dirección de memoria, lo que no es muy útil:

```
e1
#> <environment: 0x55c707c894d8>
```

En su lugar, usaremos `env_print()` que nos brinda un poco más de información:

```
env_print(e1)
#> <environment: 0x55c707c894d8>
#> Parent: <environment: global>
#> Bindings:
#> • a: <lgl>
#> • b: <chr>
#> • c: <dbl>
#> • d: <env>
```

Puede usar `env_names()` para obtener un vector de caracteres que proporcione los enlaces actuales

```
env_names(e1)
#> [1] "a" "b" "c" "d"
```

En R 3.2.0 y versiones posteriores, use `names()` para enumerar los enlaces en un entorno. Si su código necesita funcionar con R 3.1.0 o anterior, use `ls()`, pero tenga en cuenta que deberá configurar `all.names = TRUE` para mostrar todos los enlaces.

7.2.2. Entornos importantes

Hablaremos en detalle sobre entornos especiales en Section 7.4, pero por ahora necesitamos mencionar dos. El entorno actual, o `current_env()`

es el entorno en el que el código se está ejecutando actualmente. Cuando estás experimentando de forma interactiva, ese suele ser el entorno global, o `global_env()`. El entorno global a veces se llama su "área de trabajo", ya que es donde se lleva a cabo todo el cálculo interactivo (es decir, fuera de una función).

Para comparar entornos, debe usar `identical()` y no `==`. Esto se debe a que `==` es un operador vectorizado y los entornos no son vectores.

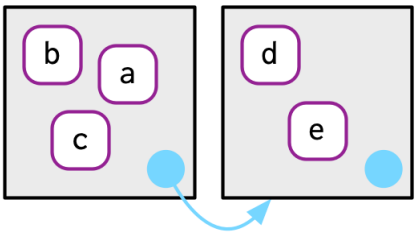
```
identical(global_env(), current_env())
#> [1] TRUE

global_env() == current_env()
#> Error in global_env() == current_env(): comparison (==) is possible only i
```

Accede al entorno global con `globalenv()` y al entorno actual con `environment()`. El entorno global se imprime como `R_GlobalEnv` y `.GlobalEnv`.

7.2.3. Padres

Cada entorno tiene un **padre**, otro entorno. En los diagramas, el padre se muestra como un pequeño círculo azul pálido y una flecha que apunta a otro entorno. El padre es lo que se usa para implementar el scoping léxico: si un nombre no se encuentra en un entorno, entonces R buscará en su padre (y así sucesivamente). Puede configurar el entorno principal proporcionando un argumento sin nombre a `env()`. Si no lo proporciona, el valor predeterminado es el entorno actual. En el siguiente código, `e2a` es el padre de `e2b`.

```
e2a <- env(d = 4, e = 5)
e2b <- env(e2a, a = 1, b = 2, c = 3)
```



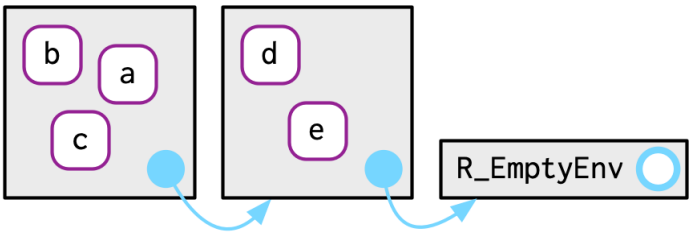
Para ahorrar espacio, normalmente no dibujaré a todos los antepasados; solo recuerda que cada vez que veas un círculo azul pálido, hay un entorno principal en alguna parte.

Puedes encontrar el padre de un entorno con `env_parent()`:

```
env_parent(e2b)
#> <environment: 0x55c706a290b8>
env_parent(e2a)
#> <environment: R_GlobalEnv>
```

Solo un entorno no tiene un padre: el entorno **vacío**. Dibujo el entorno vacío con un entorno principal vacío y, cuando el espacio lo permita, lo etiquetaré con `R_EmptyEnv`, el nombre que usa R.

```
e2c <- env(empty_env(), d = 4, e = 5)
e2d <- env(e2c, a = 1, b = 2, c = 3)
```



Los ancestros de cada ambiente eventualmente terminan con el ambiente vacío. Puedes ver todos los ancestros con `env_parents()`:

```
env_parents(e2b)
#> [[1]]   <env: 0x55c706a290b8>
#> [[2]] $ <env: global>
env_parents(e2d)
#> [[1]]   <env: 0x55c707a076e8>
#> [[2]] $ <env: empty>
```

Por defecto, `env_parents()` se detiene cuando llega al entorno global. Esto es útil porque los ancestros del entorno global incluyen todos los paquetes adjuntos, que puede ver si anula el comportamiento predeterminado como se muestra a continuación. Volveremos a estos entornos en la Section 7.4.1.

```
env_parents(e2b, last = empty_env())
#>  [[1]]   <env: 0x55c706a290b8>
#>  [[2]] $ <env: global>
#>  [[3]] $ <env: package:rlang>
#>  [[4]] $ <env: package:stats>
#>  [[5]] $ <env: package:graphics>
#>  [[6]] $ <env: package:grDevices>
#>  [[7]] $ <env: package:datasets>
#>  [[8]] $ <env: renv:shims>
#>  [[9]] $ <env: package:utils>
#> [[10]] $ <env: package:methods>
#> [[11]] $ <env: Autoloads>
#> [[12]] $ <env: package:base>
#> [[13]] $ <env: empty>
```

Use `parent.env()` para encontrar el padre de un entorno. Ninguna función base devuelve todos los ancestros.

7.2.4. Asignación superior, <<-

Los ancestros de un entorno tienen una relación importante con `<<-`. La asignación regular, `<-`, siempre crea una variable en el entorno actual. La súper asignación, `<<-`, nunca crea una variable en el entorno actual, sino que modifica una variable existente que se encuentra en un entorno principal.

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

Si `<<-` no encuentra una variable existente, creará una en el entorno global. Esto generalmente no es deseable, porque las variables globales introducen dependencias no obvias entre funciones. `<<-` se usa más a menudo junto con una fábrica de funciones, como se describe en la Section 10.2.4.

7.2.5. Conseguir y configurar

Puede obtener y establecer elementos de un entorno con `$` y `[[` de la misma manera que una lista:

```
e3 <- env(x = 1, y = 2)
e3$x
#> [1] 1
e3$z <- 3
e3[["z"]]
#> [1] 3
```

Pero no puedes usar `[[` con índices numéricos, y no puedes usar `[`:

```
e3[[1]]
#> Error in e3[[1]]: wrong arguments for subsetting an environment

e3[c("x", "y")]
#> Error in e3[c("x", "y")]: object of type 'environment' is not subsettable
```

`$` y `[[` devolverán `NULL` si el enlace no existe. Usa `env_get()` si quieres un error:

```
e3$xyz
#> NULL

env_get(e3, "xyz")
#> Error in `env_get()`:
#> ! Can't find `xyz` in environment.
```

Si desea usar un valor predeterminado si el enlace no existe, puede usar el argumento `default`.

```
env_get(e3, "xyz", default = NA)
#> [1] NA
```

Hay otras dos formas de agregar enlaces a un entorno:

- `env_poke()`[1] toma un nombre (como cadena) y un valor:

[1] Quizás se pregunte por qué rlang tiene `env_poke()` en lugar de `env_set()`. Esto es por coherencia: las funciones `_set()` devuelven una copia modificada; Las funciones `_poke()` se modifican en su lugar.

```
env_poke(e3, "a", 100)
e3$a
#> [1] 100
```

- `env_bind()` le permite vincular múltiples valores:

```
env_bind(e3, a = 10, b = 20)
env_names(e3)
#> [1] "x" "y" "z" "a" "b"
```

Puede determinar si un entorno tiene un enlace con `env_has()`:

```
env_has(e3, "a")
#>    a
#> TRUE
```

A diferencia de las listas, establecer un elemento en `NULL` no lo elimina, porque a veces desea un nombre que se refiera a `NULL`. En su lugar, usa `env_unbind()`:

```
e3$a <- NULL
env_has(e3, "a")
#>    a
#> TRUE

env_unbind(e3, "a")
env_has(e3, "a")
#>     a
#> FALSE
```

Desvincular un nombre no elimina el objeto. Ese es el trabajo del recolector de basura, que elimina automáticamente los objetos sin nombres vinculados a ellos. Este proceso se describe con más detalle en la Section 2.6.

Consulte `get()`, `assign()`, `exists()` y `rm()`. Estos están diseñados de forma interactiva para su uso con el entorno actual, por lo que trabajar con otros entornos es un poco complicado. También tenga cuidado con el argumento `inherits`: por defecto es `TRUE`, lo que significa que los equivalentes base inspeccionarán el entorno suministrado y todos sus ancestros.

7.2.6. Enlaces avanzados

Hay dos variantes más exóticas de `env_bind()`:

- `env_bind_lazy()` crea **enlaces retrasados**, que se evalúan la primera vez que se accede a ellos. Detrás de escena, los enlaces retrasados crean promesas, por lo que se comportan de la misma manera que los argumentos de función.

```
env_bind_lazy(current_env(), b = {Sys.sleep(1); 1})

system.time(print(b))
#> [1] 1
#>    user  system elapsed
#>       0       0       1
system.time(print(b))
#> [1] 1
#>    user  system elapsed
#>       0       0       0
```

  El uso principal de los enlaces retrasados es `autoload()`, que permite que los paquetes de R proporcionen conjuntos de datos que se comportan como si estuvieran cargados en la memoria, aunque solo se cargan desde el disco cuando es necesario.

- `env_bind_active()` crea **enlaces activos** que se vuelven a calcular cada vez que se accede a ellos:

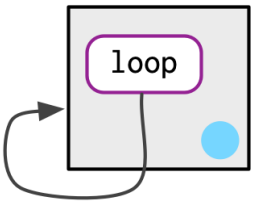
```
env_bind_active(current_env(), z1 = function(val) runif(1))

z1
#> [1] 0.0808
z1
#> [1] 0.834
```

Los enlaces activos se utilizan para implementar los campos activos de R6, sobre los que aprenderá en la Section 14.3.2.

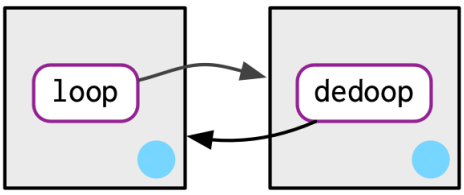
Consulte `?delayedAssign()` y `?makeActiveBinding()`.

7.2.7. Ejercicios

1. Enumera tres formas en las que un entorno difiere de una lista.
2. Cree un entorno como el que se ilustra en esta imagen.



3. Cree un par de ambientes como se ilustra en esta imagen.



4. Explique por qué `e[[1]]` y `e[c("a", "b")]` no tienen sentido cuando `e` es un entorno.

5. Cree una versión de `env_poke()` que solo vinculará nombres nuevos, nunca volverá a vincular nombres antiguos. Algunos lenguajes de programación solo hacen esto y se conocen como [lenguajes de asignación única] (http://en.wikipedia.org/wiki/Assignment_(computer_science)#Single_

6. ¿Qué hace esta función? ¿En qué se diferencia de `<<-` y por qué podría preferirlo?

```
rebind <- function(name, value, env = caller_env()) {
  if (identical(env, empty_env())) {
    stop("Can't find `", name, "`", call. = FALSE)
  } else if (env_has(env, name)) {
    env_poke(env, name, value)
  } else {
    rebind(name, value, env_parent(env))
  }
}
rebind("a", 10)
#> Error: Can't find `a`
a <- 5
rebind("a", 10)
a
#> [1] 10
```

7.3. Recursing sobre entornos

Si desea operar en todos los ancestros de un entorno, a menudo es conveniente escribir una función recursiva. Esta sección le muestra cómo, aplicando su nuevo conocimiento de entornos para escribir una función que, dado un nombre, encuentra el entorno `where()` está definido ese nombre, utilizando las reglas de alcance habituales de R.

La definición de `where()` es sencilla. Tiene dos argumentos: el nombre a buscar (como una cadena) y el entorno en el que iniciar la búsqueda. (Aprenderemos por qué `caller_env()` es un buen valor predeterminado en la Section 7.5.)

```
where <- function(name, env = caller_env()) {
  if (identical(env, empty_env())) {
    # caso base
    stop("Can't find ", name, call. = FALSE)
  } else if (env_has(env, name)) {
    # caso de exitoso
    env
  } else {
    # caso recursivo
    where(name, env_parent(env))
  }
}
```

Hay tres casos:

- El caso base: hemos llegado al entorno vacío y no hemos encontrado el enlace. No podemos ir más lejos, por lo que lanzamos un error.
- El caso exitoso: el nombre existe en este entorno, por lo que devolvemos el entorno.
- El caso recursivo: el nombre no se encontró en este entorno, así que pruebe con el padre.

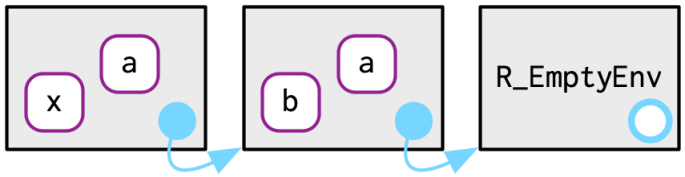
Estos tres casos se ilustran con estos tres ejemplos:

```
where("yyy")
#> Error: Can't find yyy

x <- 5
```

```
where("x")
#> <environment: R_GlobalEnv>

where("mean")
#> <environment: base>
```

Podría ayudar ver una imagen. Imagine que tiene dos entornos, como en el siguiente código y diagrama:

```
e4a <- env(empty_env(), a = 1, b = 2)
e4b <- env(e4a, x = 10, a = 11)
```



- `where("a", e4b)` encontrará `a` en `e4b`.
- `where("b", e4b)` no encuentra `b` en `e4b`, así que busca en su padre, `e4a`, y lo encuentra ahí.
- `where("c", e4b)` busca en `e4b`, entonces `e4a`, luego llega al entorno vacío y arroja un error.

Es natural trabajar con entornos recursivamente, por lo que `where()` proporciona una plantilla útil. Eliminar los detalles de `where()` muestra la estructura más claramente:

```
f <- function(..., env = caller_env()) {
  if (identical(env, empty_env())) {
    # caso base
```

```
  } else if (success) {
    # caso exitoso
  } else {
    # caso recursivo
    f(..., env = env_parent(env))
  }
}
```

Iteración versus recursividad

Es posible usar un bucle en lugar de recursividad. Creo que es más difícil de entender que la versión recursiva, pero la incluyo porque puede resultarle más fácil ver lo que sucede si no ha escrito muchas funciones recursivas.

```
f2 <- function(..., env = caller_env()) {
  while (!identical(env, empty_env())) {
    if (success) {
      # caso exitoso
      return()
    }
    # inspeccionar padre
    env <- env_parent(env)
  }

  # caso base
}
```

7.3.1. Ejercicios

1. Modifique `where()` para devolver *todos* los entornos que contienen un enlace para `name`. Piensa detenidamente qué tipo de objeto necesitará devolver la función.

2. Escribe una función llamada `fget()` que encuentre solo objetos de función. Debe tener dos argumentos, `name` y `env`, y debe obedecer las reglas regulares de alcance de las funciones: si hay un objeto con un nombre coincidente que no es una función, busque en el padre. Para un desafío adicional, agregue también un argumento `inherits` que controle si la función recurre a los padres o solo busca en un entorno.

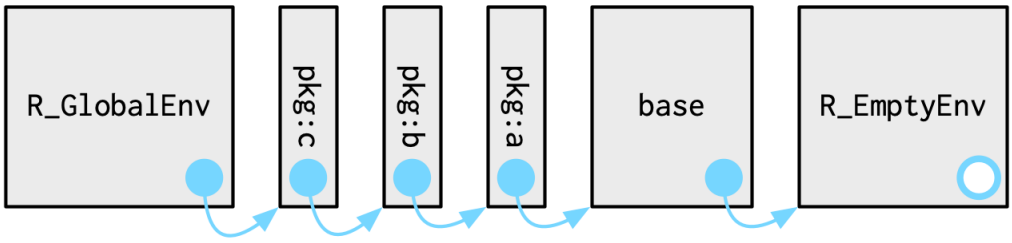
7.4. Entornos especiales

La mayoría de los entornos no los crea usted (por ejemplo, con `env()`), sino que los crea R. En esta sección, aprenderá sobre los entornos más importantes, comenzando con los entornos de paquete. Luego, aprenderá sobre el entorno de la función vinculado a la función cuando se crea y el entorno de ejecución (generalmente) efímero que se crea cada vez que se llama a la función. Finalmente, verá cómo los entornos de funciones y paquetes interactúan para admitir espacios de nombres, lo que garantiza que un paquete siempre se comporte de la misma manera, independientemente de qué otros paquetes haya cargado el usuario.

7.4.1. Entornos de paquetes y la ruta de búsqueda

Cada paquete adjunto por `library()` o `require()` se convierte en uno de los padres del entorno global. El padre inmediato del entorno global es el último paquete que adjuntó [2], el padre de ese paquete es el penúltimo paquete que adjuntó, …

[2] Tenga en cuenta la diferencia entre adjunto y cargado. Un paquete se carga automáticamente si accede a una de sus funciones usando `::`; solo se **adjunta** a la ruta de búsqueda mediante `library()` o `require()`.



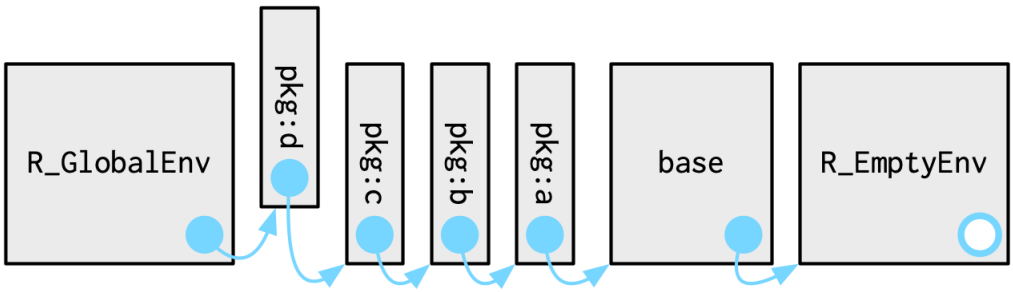
Si sigue a todos los padres hacia atrás, verá el orden en que se adjuntó cada paquete. Esto se conoce como **ruta de búsqueda** porque todos los objetos en estos entornos se pueden encontrar desde el espacio de trabajo interactivo de nivel superior. Puede ver los nombres de estos entornos con `base::search()`, o los propios entornos con `rlang::search_envs()`:

```
search()
#>  [1] ".GlobalEnv"        "package:rlang"     "package:stats"
#>  [4] "package:graphics"  "package:grDevices" "package:datasets"
#>  [7] "renv:shims"        "package:utils"     "package:methods"
#> [10] "Autoloads"         "package:base"

search_envs()
#>  [[1]] $ <env: global>
#>  [[2]] $ <env: package:rlang>
#>  [[3]] $ <env: package:stats>
#>  [[4]] $ <env: package:graphics>
#>  [[5]] $ <env: package:grDevices>
#>  [[6]] $ <env: package:datasets>
#>  [[7]] $ <env: renv:shims>
#>  [[8]] $ <env: package:utils>
#>  [[9]] $ <env: package:methods>
#> [[10]] $ <env: Autoloads>
#> [[11]] $ <env: package:base>
```

Los dos últimos entornos en la ruta de búsqueda son siempre los mismos:

- El entorno `Autoloads` utiliza enlaces retrasados para ahorrar memoria al cargar solo objetos del paquete (como grandes conjuntos de datos) cuando es necesario.
- El entorno base, `package: base` o, a veces, simplemente `base`, es el entorno del paquete base. Es especial porque debe poder iniciar la carga de todos los demás paquetes. Puedes acceder a él directamente con `base_env()`.

Tenga en cuenta que cuando adjunta otro paquete con `library()`, el entorno principal del entorno global cambia:



7.4.2. El entorno funcional

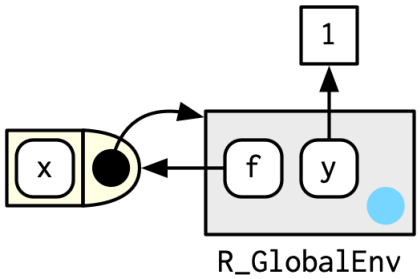
Una función vincula el entorno actual cuando se crea. Esto se denomina **entorno de función** y se utiliza para el scoping léxico. En todos los lenguajes informáticos, las funciones que capturan (o encierran) sus entornos se denominan **cierres**, razón por la cual este término a menudo se usa indistintamente con *función* en la documentación de R.

Puede obtener el entorno de la función con `fn_env()`:

```
y <- 1
f <- function(x) x + y
fn_env(f)
#> <environment: R_GlobalEnv>
```

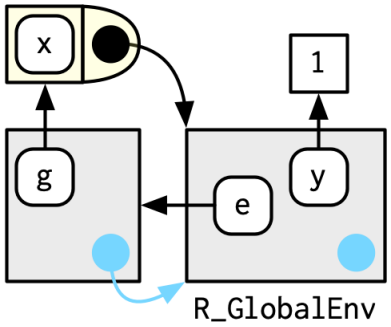
Utilice `environment(f)` para acceder al entorno de la función `f`.

En los diagramas, dibujaré una función como un rectángulo con un extremo redondeado que une un entorno.



En este caso, `f()` vincula el entorno que vincula el nombre `f` a la función. Pero ese no es siempre el caso: en el siguiente ejemplo, `g` está enlazado en un nuevo entorno `e`, pero `g()` enlaza el entorno global. La distinción entre atar y ser atado por es sutil pero importante; la diferencia es cómo encontramos `g` versus cómo `g` encuentra sus variables.

```
e <- env()
e$g <- function() 1
```



7.4.3. Espacios de nombres

En el diagrama anterior, vio que el entorno principal de un paquete varía según los otros paquetes que se hayan cargado. Esto parece preocupante: ¿no significa eso que el paquete encontrará diferentes funciones si los paquetes se cargan en un orden diferente? El objetivo de los **espacios de nombres** es asegurarse de que esto no suceda y de que todos los paquetes funcionen de la misma manera, independientemente de los paquetes que adjunte el usuario.

Por ejemplo, tome `sd()`:

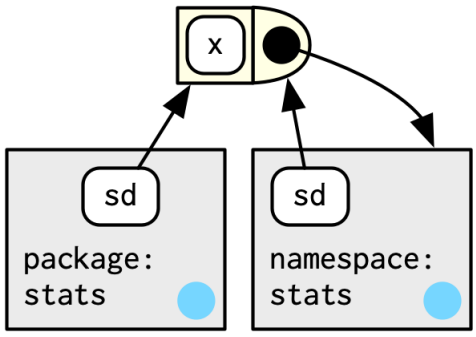
```
sd
#> function (x, na.rm = FALSE)
#> sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
#>     na.rm = na.rm))
#> <bytecode: 0x55c707adf808>
#> <environment: namespace:stats>
```

`sd()` se define en términos de `var()`, por lo que podría preocuparse de que el resultado de `sd()` se vea afectado por cualquier función llamada `var()` ya sea en el entorno global o en uno de los otros paquetes adjuntos

. R evita este problema aprovechando el entorno de función versus enlace descrito anteriormente. Cada función en un paquete está asociada con un par de entornos: el entorno del paquete, del que aprendió anteriormente, y el entorno del **espacio de nombres**.

- El entorno del paquete es la interfaz externa del paquete. Así es como usted, el usuario de R, encuentra una función en un paquete adjunto o con `::`. Su padre está determinado por la ruta de búsqueda, es decir, el orden en que se han adjuntado los paquetes.
- El entorno del espacio de nombres es la interfaz interna del paquete. El entorno del paquete controla cómo encontramos la función; el espacio de nombres controla cómo la función encuentra sus variables.

Cada enlace en el entorno del paquete también se encuentra en el entorno del espacio de nombres; esto asegura que cada función pueda usar cualquier otra función en el paquete. Pero algunos enlaces solo ocurren en el entorno del espacio de nombres. Estos se conocen como objetos internos o no exportados, que permiten ocultar al usuario detalles de implementación internos.

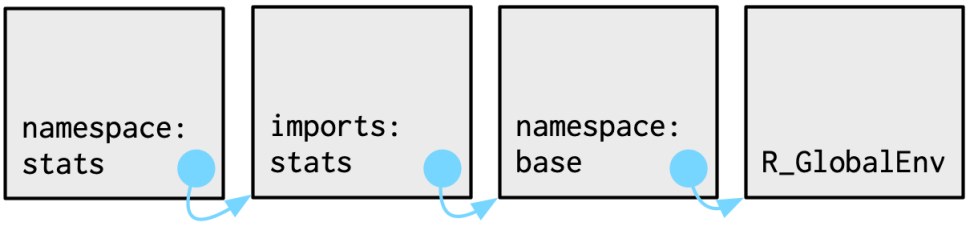


Cada entorno de espacio de nombres tiene el mismo conjunto de ancestros:

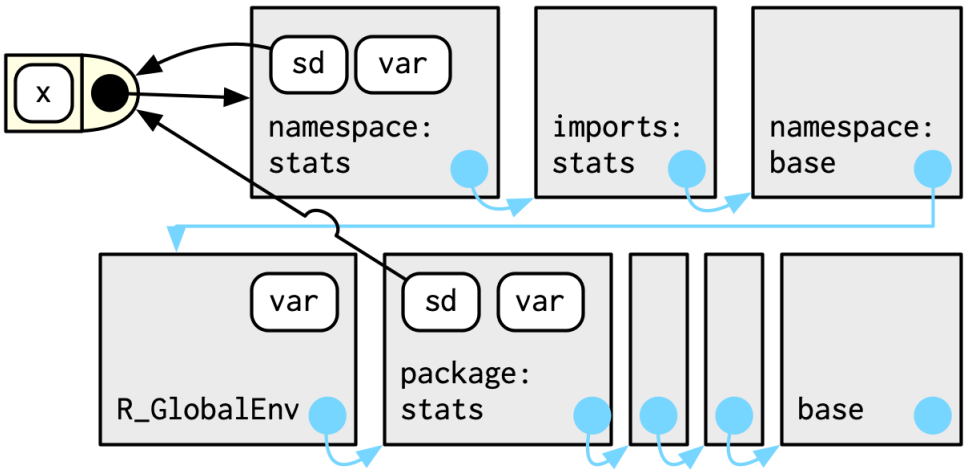
- Cada espacio de nombres tiene un entorno de **importaciones** que contiene enlaces a todas las funciones utilizadas por el paquete. El entorno de importación está controlado por el desarrollador del paquete con el archivo `NAMESPACE`.

- La importación explícita de cada función base sería tediosa, por lo que el padre del entorno de importación es el **espacio de nombres** base. El espacio de nombres base contiene los mismos enlaces que el entorno base, pero tiene un padre diferente.

- El padre del espacio de nombres base es el entorno global. Esto significa que si un enlace no está definido en el entorno de importación, el paquete lo buscará de la forma habitual. Esto suele ser una mala idea (porque hace que el código dependa de otros paquetes cargados), por lo que `R CMD check` advierte automáticamente sobre dicho código. Es necesario principalmente por razones históricas, particularmente debido a cómo funciona el envío del método S3.



Juntando todos estos diagramas obtenemos:



Entonces, cuando `sd()` busca el valor de `var`, siempre lo encuentra en una secuencia de entornos determinada por el desarrollador del paquete, pero no por el usuario del paquete. Esto garantiza que el código del paquete siempre funcione de la misma manera, independientemente de los paquetes que haya adjuntado el usuario.

No existe un vínculo directo entre el paquete y los entornos de espacio de nombres; el enlace está definido por los entornos de función.

7.4.4. Entornos de ejecución

El último tema importante que debemos cubrir es el entorno de **ejecución**. ¿Qué devolverá la siguiente función la primera vez que se ejecute? ¿Qué pasa con el segundo?

```
g <- function(x) {
  if (!env_has(current_env(), "a")) {
    message("Defining a")
```

```
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
```

Piénsalo un momento antes de seguir leyendo.

```
g(10)
#> Defining a
#> [1] 1
g(10)
#> Defining a
#> [1] 1
```

Esta función devuelve el mismo valor cada vez debido al principio de nuevo comienzo, descrito en la Section 6.4.3. Cada vez que se llama a una función, se crea un nuevo entorno para albergar la ejecución. Esto se denomina entorno de ejecución y su padre es el entorno de funciones. Ilustremos ese proceso con una función más simple. La figura Figure 7.1 ilustra las convenciones gráficas: dibujo entornos de ejecución con un padre indirecto; el entorno principal se encuentra a través del entorno de función.

```
h <- function(x) {
  # 1.
  a <- 2 # 2.
  x + a
}
y <- h(1) # 3.
```

Un entorno de ejecución suele ser efímero; una vez que la función se haya completado, el entorno se recolectará como basura. Hay varias maneras

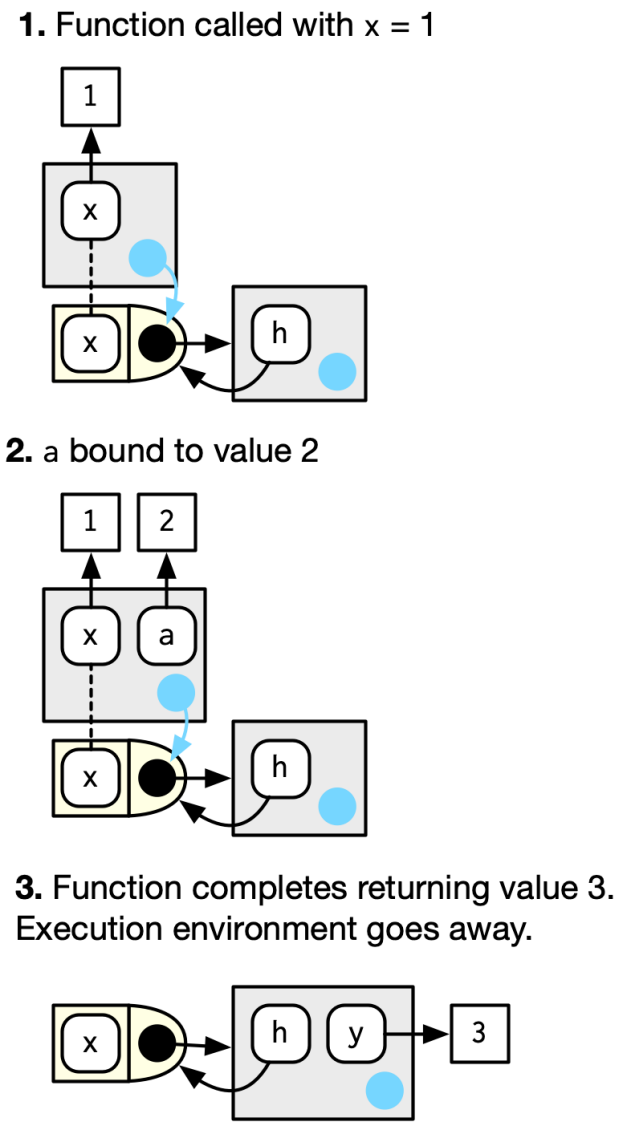


Figure 7.1.: The execution environment of a simple function call. Note that the parent of the execution environment is the function environment.

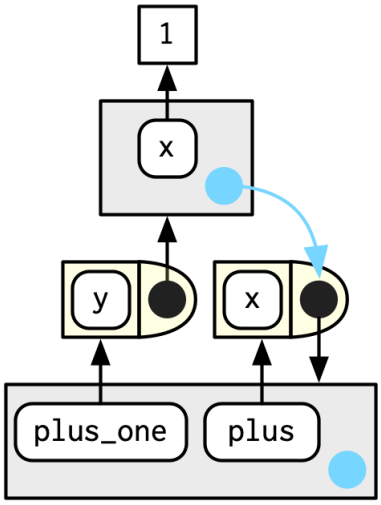
de hacer que se quede por más tiempo. El primero es devolverlo explícitamente:

```
h2 <- function(x) {
  a <- x * 2
  current_env()
}

e <- h2(x = 10)
env_print(e)
#> <environment: 0x55c70a9c58d0>
#> Parent: <environment: global>
#> Bindings:
#> • a: <dbl>
#> • x: <dbl>
fn_env(h2)
#> <environment: R_GlobalEnv>
```

Otra forma de capturarlo es devolver un objeto con un enlace a ese entorno, como una función. El siguiente ejemplo ilustra esa idea con una fábrica de funciones, `plus()`. Usamos esa fábrica para crear una función llamada `plus_one()`.

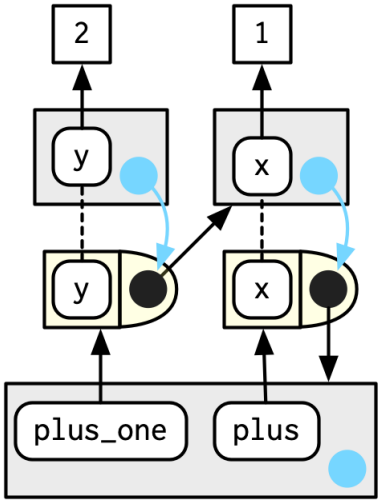
Están sucediendo muchas cosas en el diagrama porque el entorno envolvente de `plus_one()` es el entorno de ejecución de `plus()`.

```
plus <- function(x) {
  function(y) x + y
}

plus_one <- plus(1)
plus_one
#> function(y) x + y
#> <environment: 0x55c70b7c9cd0>
```



¿Qué sucede cuando llamamos `plus_one()`? Su entorno de ejecución tendrá el entorno de ejecución capturado de `plus()` como padre:

```
plus_one(2)
#> [1] 3
```



Aprenderá más sobre las fábricas de funciones en la Section 10.2.

7.4.5. Ejercicios

1. ¿En qué se diferencia `search_envs()` de `env_parents(global_env())`?
2. Dibuje un diagrama que muestre los entornos circundantes de esta función:

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

```
}
f1(1)
```

3. Escriba una versión mejorada de `str()` que proporcione más información sobre las funciones. Muestre dónde se encontró la función y en qué entorno se definió.

7.5. Pilas de llamadas

Hay un último entorno que debemos explicar, el entorno **caller**, al que se accede con `rlang::caller_env()`. Esto proporciona el entorno desde el que se llamó a la función y, por lo tanto, varía en función de cómo se llame a la función, no de cómo se creó. Como vimos anteriormente, este es un valor predeterminado útil cada vez que escribe una función que toma un entorno como argumento.

`parent.frame()` es equivalente a `caller_env()`; solo tenga en cuenta que devuelve un entorno, no un marco.

Para comprender completamente el entorno de la persona que llama, debemos analizar dos conceptos relacionados: la **pila de llamadas**, que se compone de **marcos**. La ejecución de una función crea dos tipos de contexto. Ya aprendió sobre uno: el entorno de ejecución es un elemento secundario del entorno de función, que está determinado por el lugar donde se creó la función. Hay otro tipo de contexto creado por donde se llamó a la función: esto se llama la pila de llamadas.

7.5.1. Pilas de llamadas simples

Ilustremos esto con una secuencia simple de llamadas: `f()` llama a `g()` llama a `h()`.

```
f <- function(x) {
  g(x = 2)
}
g <- function(x) {
  h(x = 3)
}
h <- function(x) {
  stop()
}
```

La forma más común de ver una pila de llamadas en R es mirando el `traceback()` después de que haya ocurrido un error:

```
f(x = 1)
#> Error:
traceback()
#> 4: stop()
#> 3: h(x = 3)
#> 2: g(x = 2)
#> 1: f(x = 1)
```

En lugar de `stop()` + `traceback()` para entender la pila de llamadas, vamos a usar `lobstr::cst()` para imprimir el árbol de pilas de llamadas (**c**all **s**tack *t*ree, en inglés):

```
h <- function(x) {
  lobstr::cst()
}
f(x = 1)
#>
#>  f(x = 1)
#>    g(x = 2)
```

```
#>      h(x = 3)
#>        lobstr::cst()
```

Esto nos muestra que `cst()` fue llamado desde `h()`, que fue llamado desde `g()`, que fue llamado desde `f()`. Tenga en cuenta que el orden es el opuesto de `traceback()`. A medida que las pilas de llamadas se vuelven más complicadas, creo que es más fácil entender la secuencia de llamadas si comienza desde el principio, en lugar del final (es decir, `f()` llama a `g()`; en lugar de `g()` fue llamado por `f()`).

7.5.2. Evaluación perezosa

La pila de llamadas anterior es simple: mientras obtiene una pista de que hay una estructura similar a un árbol involucrada, todo sucede en una sola rama. Esto es típico de una pila de llamadas cuando todos los argumentos se evalúan con entusiasmo.

Vamos a crear un ejemplo más complicado que implique una evaluación perezosa. Crearemos una secuencia de funciones, `a()`, `b()`, `c()`, que pasan un argumento `x`.

```
a <- function(x) b(x)
b <- function(x) c(x)
c <- function(x) x

a(f())
#>
#>  a(f())
#>    b(x)
#>      c(x)
#>  f()
#>    g(x = 2)
```

```
#>       h(x = 3)
#>         lobstr::cst()
```

`x` se evalúa perezosamente, por lo que este árbol tiene dos ramas. En la primera rama `a()` llama a `b()`, luego `b()` llama a `c()`. La segunda rama comienza cuando `c()` evalúa su argumento `x`. Este argumento se evalúa en una nueva rama porque el entorno en el que se evalúa es el entorno global, no el entorno de `c()`.

7.5.3. Marcos

Cada elemento de la pila de llamadas es un **marco**[3], también conocido como contexto de evaluación. El marco es una estructura de datos interna extremadamente importante, y el código R solo puede acceder a una pequeña parte de la estructura de datos porque manipularlo romperá R. Un marco tiene tres componentes clave:

- Una expresión (etiquetada con `expr`) que da la llamada a la función. Esto es lo que imprime `traceback()`.
- Un entorno (etiquetado con `env`), que suele ser el entorno de ejecución de una función. Hay dos excepciones principales: el entorno del marco global es el entorno global, y llamar a `eval()` también genera marcos, donde el entorno puede ser cualquier cosa.
- Un padre, la llamada anterior en la pila de llamadas (se muestra con una flecha gris).

La figura Figure 7.2 ilustra la pila para la llamada a `f(x = 1)` que se muestra en la Section 7.5.1.

[3]NB: `?environment` usa marco en un sentido diferente: "Los entornos consisten en un *marco*, o una colección de objetos con nombre, y un puntero a un entorno envolvente". Evitamos este sentido de marco, que proviene de S, porque es muy específico y no se usa mucho en la base R. Por ejemplo, el marco en `parent.frame()` es un contexto de ejecución, no una colección de objetos con nombre.

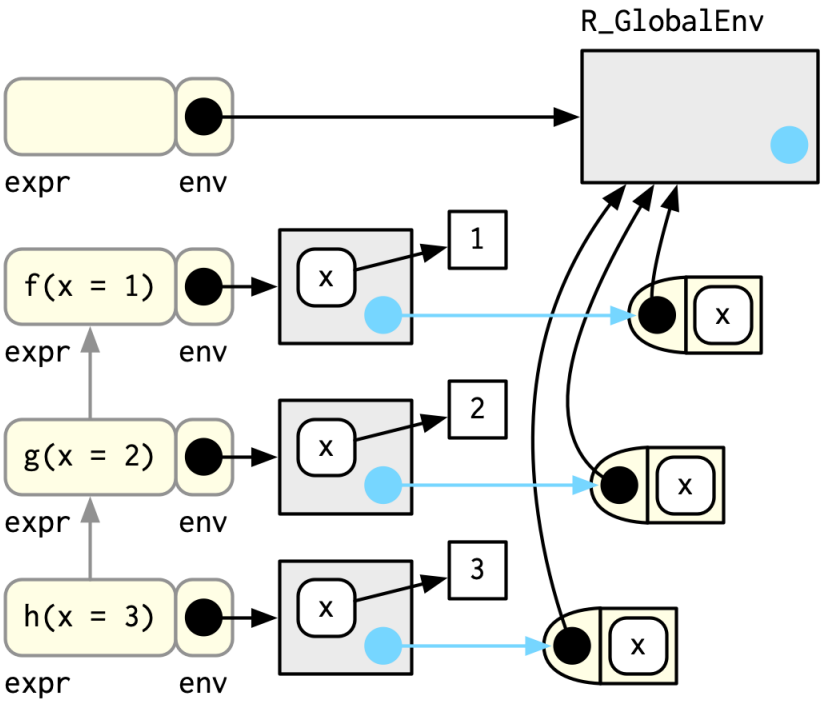


Figure 7.2.: The graphical depiction of a simple call stack

(Para centrarme en los entornos de llamada, he omitido los enlaces en el entorno global de `f`, `g` y `h` a los objetos de función respectivos.)

El marco también contiene controladores de salida creados con `on.exit()`, reinicios y controladores para el sistema de condiciones, y a qué contexto `return()` cuando se completa una función. Estos son detalles internos importantes a los que no se puede acceder con el código R.

7.5.4. Alcance dinámico

La búsqueda de variables en la pila de llamadas en lugar de en el entorno adjunto se denomina **ámbito dinámico**. Pocos lenguajes implementan el alcance dinámico (Emacs Lisp es una [excepción notable] (http://www.gnu.org/software/emacs/emacs-paper.html#SEC15).) Esto se debe a que el alcance dinámico hace que sea mucho más difícil razonar sobre cómo opera una función: no solo necesita saber cómo se definió, también necesita saber el contexto en el que se llamó. El alcance dinámico es principalmente útil para desarrollar funciones que ayudan al análisis interactivo de datos y es uno de los temas tratados en el Chapter 20.

7.5.5. Ejercicios

1. Escriba una función que enumere todas las variables definidas en el entorno en el que se llamó. Debería devolver los mismos resultados que `ls()`.

7.6. Como estructuras de datos

Además de potenciar el alcance, los entornos también son estructuras de datos útiles por derecho propio porque tienen semántica de referencia. Hay tres problemas comunes que pueden ayudar a resolver:

- **Evitar copias de datos de gran tamaño**. Dado que los entornos tienen semántica de referencia, nunca creará una copia accidentalmente. Pero es complicado trabajar con entornos desnudos, por lo que en su lugar recomiendo usar objetos R6, que se construyen sobre los entornos. Obtenga más información en el Chapter 14.

- **Administrar el estado dentro de un paquete**. Los entornos explícitos son útiles en los paquetes porque le permiten mantener el estado en las llamadas a funciones. Normalmente, los objetos de un paquete están bloqueados, por lo que no puede modificarlos directamente. En su lugar, puedes hacer algo como esto:

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

get_a <- function() {
  my_env$a
}
set_a <- function(value) {
  old <- my_env$a
  my_env$a <- value
  invisible(old)
}
```

  Devolver el valor anterior de las funciones de establecimiento es un buen patrón porque hace que sea más fácil restablecer el valor anterior junto con `on.exit()` (Section 6.7.4).

- **Como hashmap**. Un hashmap es una estructura de datos que toma tiempo constante, O(1), para encontrar un objeto basado en su nombre. Los entornos proporcionan este comportamiento de forma predeterminada, por lo que se pueden usar para simular un mapa hash. Vea el paquete hash (Brown 2013) para un desarrollo completo de esta idea.

7.7. Respuestas de la prueba

1. Hay cuatro formas: cada objeto en un entorno debe tener un nombre; el orden no importa; los ambientes tienen padres; los entornos tienen semántica de referencia.
2. El padre del entorno global es el último paquete que cargó. El único entorno que no tiene un padre es el entorno vacío.
3. El entorno envolvente de una función es el entorno donde se creó. Determina dónde una función busca variables.
4. Use `caller_env()` o `parent.frame()`.
5. `<-` siempre crea un enlace en el entorno actual; `<<-` vuelve a enlazar un nombre existente en un padre del entorno actual.

8. Condiciones

8.1. Introducción

El sistema de **condición** proporciona un conjunto emparejado de herramientas que permiten al autor de una función indicar que algo inusual está sucediendo y al usuario de esa función manejarlo. El autor de la función **señala** las condiciones con funciones como `stop()` (para errores), `warning()` (para advertencias) y `message()` (para mensajes), luego el usuario de la función puede manejarlas. con funciones como `tryCatch()` y `withCallingHandlers()`. Comprender el sistema de condiciones es importante porque a menudo necesitará desempeñar ambos roles: señalar las condiciones de las funciones que crea y manejar las condiciones señaladas por las funciones que llama.

R ofrece un sistema de condiciones muy poderoso basado en ideas de Common Lisp. Al igual que el enfoque de R para la programación orientada a objetos, es bastante diferente a los lenguajes de programación actualmente populares, por lo que es fácil malinterpretarlo y se ha escrito relativamente poco sobre cómo usarlo de manera efectiva. Históricamente, esto ha significado que pocas personas (incluido yo mismo) han aprovechado al máximo su poder. El objetivo de este capítulo es remediar esa situación. Aquí aprenderá sobre las grandes ideas del sistema de condiciones de R, además de aprender un montón de herramientas prácticas que fortalecerán su código.

Encontré dos recursos particularmente útiles al escribir este capítulo. También puede leerlos si desea obtener más información sobre las inspiraciones

y motivaciones del sistema:

- *Un prototipo de un sistema de condiciones para R* de Robert Gentleman y Luke Tierney. Esto describe una versión anterior del sistema de condiciones de R. Si bien la implementación ha cambiado un poco desde que se escribió este documento, proporciona una buena descripción general de cómo encajan las piezas y algo de motivación para su diseño.

- *Más allá del manejo de excepciones: condiciones y reinicios* de Peter Seibel. Esto describe el manejo de excepciones en Lisp, que resulta ser muy similar al enfoque de R. Proporciona una motivación útil y ejemplos más sofisticados. He proporcionado una traducción R del capítulo en http://adv-r.had.co.nz/beyond-exception-handling.html.

También encontré útil trabajar con el código C subyacente que implementa estas ideas. Si está interesado en entender cómo funciona todo, puede encontrar mis notas de utilidad.

Prueba

¿Quieres saltarte este capítulo? Anímate, si puedes responder las siguientes preguntas. Encuentre las respuestas al final del capítulo en la Section 8.7.

1. ¿Cuáles son los tres tipos de condiciones más importantes?

2. ¿Qué función utiliza para ignorar los errores en el bloque de código?

3. ¿Cuál es la principal diferencia entre `tryCatch()` y `withCallingHandlers()`?

4. ¿Por qué podría querer crear un objeto de error personalizado?

Estructura

- La Section 8.2 presenta las herramientas básicas para las condiciones de señalización y analiza cuándo es apropiado usar cada tipo.
- La Section 8.3 le enseña sobre las herramientas más simples para manejar condiciones: funciones como `try()` y `suppressMessages()` que tragan condiciones y evitan que lleguen al nivel superior.
- La Section 8.4 introduce la condición **objeto** y las dos herramientas fundamentales del manejo de condiciones: `tryCatch()` para condiciones de error y `withCallingHandlers()` para todo lo demás.
- La Section 8.5 le muestra cómo ampliar los objetos de condición incorporados para almacenar datos útiles que los controladores de condiciones pueden usar para tomar decisiones más informadas.
- La Section 8.6 cierra el capítulo con una bolsa de sorpresas de aplicaciones prácticas basadas en las herramientas de bajo nivel que se encuentran en las secciones anteriores.

8.1.1. Requisitos previos

Además de las funciones básicas de R, este capítulo utiliza funciones de señalización y manejo de condiciones de rlang.

```
library(rlang)
```

8.2. Condiciones de señalización

Hay tres condiciones que puede señalar en el código: errores, advertencias y mensajes.

- Los errores son los más graves; indican que no hay forma de que una función continúe y la ejecución debe detenerse.
- Las advertencias se encuentran un poco entre los errores y los mensajes, y generalmente indican que algo salió mal pero la función se pudo recuperar al menos parcialmente.
- Los mensajes son los más suaves; son una forma de informar a los usuarios que se ha realizado alguna acción en su nombre.

Hay una condición final que solo se puede generar de forma interactiva: una interrupción, que indica que el usuario ha interrumpido la ejecución presionando Escape, Ctrl + Pausa o Ctrl + C (según la plataforma).

Las condiciones generalmente se muestran de manera destacada, en negrita o en color rojo, según la interfaz de R. Puede distinguirlos porque los errores siempre comienzan con "Error", las advertencias con "Warning" o "Warning message" y los mensajes sin nada.

```
stop("Así es como se ve un error")
#> Error in eval(expr, envir, enclos): Así es como se ve un error

warning("Así es como se ve una advertencia")
#> Warning: Así es como se ve una advertencia

message("Así es como se ve un mensaje")
#> Así es como se ve un mensaje
```

Las siguientes tres secciones describen errores, advertencias y mensajes con más detalle.

8.2.1. Errores

En base R, los errores son señalados, o **lanzados**, por `stop()`:

```
f <- function() g()
g <- function() h()
h <- function() stop("¡Esto es un error!")

f()
#> Error in h(): ¡Esto es un error!
```

De forma predeterminada, el mensaje de error incluye la llamada, pero esto normalmente no es útil (y recapitula información que puede obtener fácilmente de `traceback()`), por lo que creo que es una buena práctica usar `call. = FALSE`[1]:

```
h <- function() stop("¡Esto es un error!", call. = FALSE)
f()
#> Error: ¡Esto es un error!
```

El rlang equivalente a `stop()`, `rlang::abort()`, hace esto automáticamente. Usaremos `abort()` a lo largo de este capítulo, pero no llegaremos a su característica más convincente, la capacidad de agregar metadatos adicionales al objeto de condición, hasta que estemos cerca del final del capítulo.

```
h <- function() abort("This is an error!")
f()
#> Error in `h()`:
#> ! This is an error!
```

(NB: `stop()` pega varias entradas juntas, mientras que `abort()` no lo hace. Para crear mensajes de error complejos con abortar, recomiendo usar `glue::glue()`. Esto nos permite usar otros argumentos para `abortar ()` para características útiles que aprenderá en la Section 8.5.)

[1] El final `.` en `call.` es una peculiaridad de `stop()`; no leas nada.

Los mejores mensajes de error le dicen qué está mal y le indican la dirección correcta para solucionar el problema. Escribir buenos mensajes de error es difícil porque los errores generalmente ocurren cuando el usuario tiene un modelo mental defectuoso de la función. Como desarrollador, es difícil imaginar cómo el usuario podría estar pensando incorrectamente sobre su función y, por lo tanto, es difícil escribir un mensaje que dirija al usuario en la dirección correcta. Dicho esto, la guía de estilo de tidyverse analiza algunos principios generales que hemos encontrado útiles: http://style.tidyverse.org/error-messages.html.

8.2.2. Advertencias

Las advertencias, señaladas por `warning()`, son más débiles que los errores: indican que algo salió mal, pero el código pudo recuperarse y continuar. A diferencia de los errores, puede tener múltiples advertencias de una sola llamada de función:

```
fw <- function() {
  cat("1\n")
  warning("W1")
  cat("2\n")
  warning("W2")
  cat("3\n")
  warning("W3")
}
```

De forma predeterminada, las advertencias se almacenan en caché y se imprimen solo cuando el control vuelve al nivel superior:

```
fw()
#> 1
#> 2
```

```
#> 3
#> Warning messages:
#> 1: In f() : W1
#> 2: In f() : W2
#> 3: In f() : W3
```

Puedes controlar este comportamiento con la opción `warn`:

- Para que las advertencias aparezcan inmediatamente, configure `options(warn = 1)`.
- Para convertir las advertencias en errores, establezca `options(warn = 2)`. Esta suele ser la forma más fácil de depurar una advertencia, ya que una vez que se trata de un error, puede usar herramientas como `traceback()` para encontrar la fuente.
- Restaurar el comportamiento predeterminado con `options(warn = 0)`.

Al igual que `stop()`, `warning()` también tiene un argumento de llamada. Es un poco más útil (ya que las advertencias a menudo están más lejos de su fuente), pero generalmente lo suprimo con `call. = FALSE`. Al igual que `rlang::abort()`, el equivalente en rlang de `warning()`, `rlang::warn()`, también suprime la `call` por defecto.

Las advertencias ocupan un lugar algo desafiante entre los mensajes ("debe saber sobre esto") y los errores ("¡debe arreglar esto!"), y es difícil dar consejos precisos sobre cuándo usarlos. En general, tenga cuidado, ya que es fácil pasar por alto las advertencias si hay muchos otros resultados y no desea que su función se recupere con demasiada facilidad de una entrada claramente no válida. En mi opinión, la base R tiende a abusar de las advertencias, y muchas advertencias en la base R estarían mejor como errores. Por ejemplo, creo que estas advertencias serían más útiles como errores:

```
formals(1)
#> Warning in formals(fun): argument is not a function
#> NULL

file.remove("this-file-doesn't-exist")
#> Warning in file.remove("this-file-doesn't-exist"): cannot remove
#> file 'this-file-doesn't-exist', reason 'No such file or directory'
#> [1] FALSE

lag(1:3, k = 1.5)
#> Warning in lag.default(1:3, k = 1.5): 'k' is not an integer
#> [1] 1 2 3
#> attr(,"tsp")
#> [1] -1  1  1

as.numeric(c("18", "30", "50+", "345,678"))
#> Warning: NAs introduced by coercion
#> [1] 18 30 NA NA
```

Solo hay un par de casos en los que usar una advertencia es claramente apropiado:

- Cuando **desaproba** una función, desea permitir que el código anterior continúe funcionando (por lo que ignorar la advertencia está bien), pero desea alentar al usuario a cambiar a una nueva función.
- Cuando esté razonablemente seguro de que puede solucionar un problema: si estuviera 100% seguro de que podría solucionar el problema, no necesitaría ningún mensaje; si no estuviera seguro de poder solucionar correctamente el problema, arrojaría un error.

De lo contrario, use las advertencias con moderación y considere cuidadosamente si un error sería más apropiado.

8.2.3. Mensajes

Los mensajes, señalados por `message()`, son informativos; utilícelos para decirle al usuario que ha hecho algo en su nombre. Los buenos mensajes son un acto de equilibrio: desea proporcionar la información suficiente para que el usuario sepa lo que está sucediendo, pero no tanto como para que se sienta abrumado.

Los mensajes, `message()`, se muestran inmediatamente y no tienen un argumento `call.`:

```
fm <- function() {
  cat("1\n")
  message("M1")
  cat("2\n")
  message("M2")
  cat("3\n")
  message("M3")
}

fm()
#> 1
#> M1
#> 2
#> M2
#> 3
#> M3
```

Buenos lugares para usar un mensaje son:

- Cuando un argumento predeterminado requiere una cantidad de cálculo no trivial y desea decirle al usuario qué valor se utilizó. Por ejemplo, ggplot2 informa la cantidad de contenedores utilizados si no proporciona un `binwidth`.

- En funciones que son convocadas principalmente por sus efectos secundarios que de otro modo serían silenciosos. Por ejemplo, al escribir archivos en el disco, llamar a una API web o escribir en una base de datos, es útil proporcionar mensajes de estado regulares que le informen al usuario lo que está sucediendo.
- Cuando esté a punto de iniciar un proceso de ejecución prolongada sin resultados intermedios. Una barra de progreso (por ejemplo, con progress) es mejor, pero un mensaje es un buen lugar para comenzar.
- Al escribir un paquete, a veces desea mostrar un mensaje cuando se carga su paquete (es decir, en `.onAttach()`); aquí debes usar `packageStartupMessage()`.

En general, cualquier función que produzca un mensaje debería tener alguna forma de suprimirlo, como un argumento `quiet = TRUE`. Es posible suprimir todos los mensajes con `suppressMessages()`, como aprenderá en breve, pero también es bueno dar un control más detallado.

Es importante comparar `message()` con el `cat()` estrechamente relacionado. En términos de uso y resultado, parecen bastante similares[2]:

```
cat("Hi!\n")
#> Hi!

message("Hi!")
#> Hi!
```

Sin embargo, los *propósitos* de `cat()` y `message()` son diferentes. Usa `cat()` cuando el rol principal de la función es imprimir en la consola, como los métodos `print()` o `str()`. Usa `message()` como un canal lateral para imprimir en la consola cuando el propósito principal de la función es otra cosa. En otras palabras, `cat()` es para cuando el usuario *pide* que se

[2]Pero tenga en cuenta que `cat()` requiere un final explícito `"\n"` para imprimir una nueva línea.

imprima algo y `message()` es para cuando el desarrollador *elige* imprimir algo.

8.2.4. Ejercicios

1. Escriba un contenedor alrededor de `file.remove()` que arroje un error si el archivo a eliminar no existe.

2. ¿Qué hace el argumento `appendLF` para `message()`? ¿Cómo se relaciona con `cat()`?

8.3. Ignorando las condiciones

La forma más sencilla de manejar las condiciones en R es simplemente ignorarlas:

- Ignora los errores con `try()`.
- Ignora las advertencias con `suppressWarnings()`.
- Ignorar mensajes con `suppressMessages()`.

Estas funciones son de mano dura, ya que no puede usarlas para suprimir un solo tipo de condición que conozca, mientras permite que pase todo lo demás. Volveremos a ese desafío más adelante en el capítulo.

`try()` permite que la ejecución continúe incluso después de que haya ocurrido un error. Normalmente, si ejecuta una función que arroja un error, finaliza inmediatamente y no devuelve un valor:

```
f1 <- function(x) {
  log(x)
  10
}
```

```
f1("x")
#> Error in log(x): non-numeric argument to mathematical function
```

Sin embargo, si ajusta la declaración que crea el error en `try()`, se mostrará el mensaje de error [3] pero la ejecución continuará:

```
f2 <- function(x) {
  try(log(x))
  10
}
f2("a")
#> Error in log(x) : non-numeric argument to mathematical function
#> [1] 10
```

Es posible, pero no recomendado, guardar el resultado de `try()` y realizar diferentes acciones en función de si el código tuvo éxito o no [4]. En su lugar, es mejor usar `tryCatch()` o un ayudante de nivel superior; aprenderá acerca de ellos en breve.

Un patrón simple, pero útil, es hacer una asignación dentro de la llamada: esto le permite definir un valor predeterminado que se usará si el código no funciona correctamente. Esto funciona porque el argumento se evalúa en el entorno de llamada, no dentro de la función. (Consulte la Section 6.5.1 para obtener más detalles).

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

`suppressWarnings()` y `suppressMessages()` suprimir todas las advertencias y mensajes. A diferencia de los errores, los mensajes y advertencias

[3] Puede suprimir el mensaje con `try(..., silent = TRUE)`.
[4] Puede saber si la expresión falló porque el resultado tendrá clase `try-error`.

no terminan la ejecución, por lo que puede haber múltiples advertencias y mensajes señalados en un solo bloque.

```
suppressWarnings({
  warning("Uhoh!")
  warning("Otra advertencia")
  1
})
#> [1] 1

suppressMessages({
  message("Hola")
  2
})
#> [1] 2

suppressWarnings({
  message("Todavía puedes verme")
  3
})
#> Todavía puedes verme
#> [1] 3
```

8.4. Controladores de condiciones

Cada condición tiene un comportamiento predeterminado: los errores detienen la ejecución y regresan al nivel superior, las advertencias se capturan y muestran en conjunto y los mensajes se muestran inmediatamente. Los **controladores** de condiciones nos permiten anular o complementar temporalmente el comportamiento predeterminado.

Dos funciones, `tryCatch()` y `withCallingHandlers()`, nos permiten registrar controladores, funciones que toman la condición señalada como

único argumento. Las funciones de registro tienen la misma forma básica:

```
tryCatch(
  error = function(cnd) {
    # código para ejecutar cuando se lanza un error
  },
  code_to_run_while_handlers_are_active
)

withCallingHandlers(
  warning = function(cnd) {
    # código para ejecutar cuando se señale una advertencia
  },
  message = function(cnd) {
    # código para ejecutar cuando se señala el mensaje
  },
  code_to_run_while_handlers_are_active
)
```

Se diferencian en el tipo de controladores que crean:

- `tryCatch()` define controladores **que salen**; después de manejar la condición, el control regresa al contexto donde se llamó a `tryCatch()`. Esto hace que `tryCatch()` sea más adecuado para trabajar con errores e interrupciones, ya que estos tienen que salir de todos modos.

- `withCallingHandlers()` define controladores de **llamadas**; después de capturar la condición, el control vuelve al contexto donde se señaló la condición. Esto lo hace más adecuado para trabajar con condiciones sin error.

Pero antes de que podamos aprender y usar estos controladores, necesitamos hablar un poco sobre la condición **objetos**. Estos se crean implícitamente cada vez que señala una condición, pero se vuelven explícitos dentro del controlador.

8.4.1. Objetos de condición

Hasta ahora, solo hemos señalado las condiciones y no hemos mirado los objetos que se crean detrás de escena. La forma más fácil de ver un objeto de condición es atrapar uno de una condición señalada. ese es el trabajo de `rlang::catch_cnd()`:

```
cnd <- catch_cnd(stop("An error"))
str(cnd)
#> List of 2
#>  $ message: chr "An error"
#>  $ call   : language force(expr)
#>  - attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Las condiciones integradas son listas con dos elementos:

- `menssage`, un vector de caracteres de longitud 1 que contiene el texto para mostrar a un usuario. Para extraer el mensaje, utilice `conditionMessage(cnd)`.

- `call`, la llamada que activó la condición. Como se describió anteriormente, no usamos la llamada, por lo que a menudo será `NULL`. Para extraerlo, usa `conditionCall(cnd)`.

Las condiciones personalizadas pueden contener otros componentes, que analizaremos en la Section 8.5.

Las condiciones también tienen un atributo `class`, lo que las convierte en objetos S3. No hablaremos de S3 hasta el Chapter 13, pero afortunadamente, incluso si no conoce S3, los objetos de condición son bastante simples. Lo más importante que debe saber es que el atributo `class` es un vector de caracteres y determina qué controladores coincidirán con la condición.

8.4.2. Controladores de salida

`tryCatch()` egistra los controladores existentes y, por lo general, se utiliza para controlar las condiciones de error. Le permite anular el comportamiento de error predeterminado. Por ejemplo, el siguiente código devolverá `NA` en lugar de arrojar un error:

```
f3 <- function(x) {
  tryCatch(
    error = function(cnd) NA,
    log(x)
  )
}

f3("x")
#> [1] NA
```

Si no se señalan condiciones, o si la clase de la condición señalada no coincide con el nombre del controlador, el código se ejecuta normalmente:

```
tryCatch(
  error = function(cnd) 10,
  1 + 1
)
#> [1] 2
```

```
tryCatch(
  error = function(cnd) 10,
  {
    message("Hi!")
    1 + 1
  }
)
#> Hi!
#> [1] 2
```

Los controladores configurados por `tryCatch()` se denominan controladores **exiting** porque después de señalar la condición, el control pasa al controlador y nunca vuelve al código original, lo que significa que el código sale:

```
tryCatch(
  message = function(cnd) "There",
  {
    message("Here")
    stop("This code is never run!")
  }
)
#> [1] "There"
```

El código protegido se evalúa en el entorno de `tryCatch()`, pero el código del controlador no, porque los controladores son funciones. Es importante recordar esto si está tratando de modificar objetos en el entorno principal.

Las funciones del controlador se llaman con un solo argumento, el objeto de condición. Llamo a este argumento `cnd`, por convención. Este valor es solo moderadamente útil para las condiciones base porque contienen

relativamente pocos datos. Es más útil cuando crea sus propias condiciones personalizadas, como verá en breve.

```
tryCatch(
  error = function(cnd) {
    paste0("--", conditionMessage(cnd), "--")
  },
  stop("This is an error")
)
#> [1] "--This is an error--"
```

`tryCatch()` tiene otro argumento: `finally`. Especifica un bloque de código (no una función) para ejecutar independientemente de si la expresión inicial tiene éxito o falla. Esto puede ser útil para la limpieza, como eliminar archivos o cerrar conexiones. Esto es funcionalmente equivalente a usar `on.exit()` (y de hecho así es como se implementa), pero puede envolver fragmentos de código más pequeños que una función completa.

```
path <- tempfile()
tryCatch(
  {
    writeLines("Hi!", path)
    # ...
  },
  finally = {
    # always run
    unlink(path)
  }
)
```

8.4.3. Controladores de llamadas

Los controladores configurados por `tryCatch()` se denominan controladores de salida porque hacen que el código se cierre una vez que se ha detectado la condición. Por el contrario, `withCallingHandlers()` configura controladores de **llamadas**: la ejecución del código continúa normalmente una vez que el controlador regresa. Esto tiende a hacer que `withCallingHandlers()` sea un emparejamiento más natural con las condiciones sin error. Los controladores de salida y llamada usan "controlador" en sentidos ligeramente diferentes:

- Un controlador existente maneja una señal como tú manejas un problema; hace que el problema desaparezca.
- Un controlador de llamadas maneja una señal como usted maneja un automóvil; el coche todavía existe.

Compara los resultados de `tryCatch()` y `withCallingHandlers()` en el siguiente ejemplo. Los mensajes no se imprimen en el primer caso, porque el código finaliza una vez que se completa el controlador de salida. Se imprimen en el segundo caso, porque un controlador de llamadas no sale.

```
tryCatch(
  message = function(cnd) cat("Caught a message!\n"),
  {
    message("Someone there?")
    message("Why, yes!")
  }
)
#> Caught a message!

withCallingHandlers(
  message = function(cnd) cat("Caught a message!\n"),
  {
```

```
    message("Someone there?")
    message("Why, yes!")
  }
)
#> Caught a message!
#> Someone there?
#> Caught a message!
#> Why, yes!
```

Los controladores se aplican en orden, por lo que no debe preocuparse por quedar atrapado en un bucle infinito. En el siguiente ejemplo, el `message()` señalado por el controlador tampoco queda atrapado:

```
withCallingHandlers(
  message = function(cnd) message("Second message"),
  message("First message")
)
#> Second message
#> First message
```

(Pero tenga cuidado si tiene varios controladores y algunos controladores señalan condiciones que podrían ser capturadas por otro controlador: tendrá que pensar detenidamente en la orden).

El valor de retorno de un controlador de llamadas se ignora porque el código continúa ejecutándose después de que se completa el controlador; ¿Adónde iría el valor de retorno? Eso significa que los controladores de llamadas solo son útiles por sus efectos secundarios.

Un efecto secundario importante exclusivo de los controladores de llamadas es la capacidad de **amortiguar** la señal. De forma predeterminada, una condición continuará propagándose a los controladores principales, hasta el controlador predeterminado (o un controlador existente, si se proporciona):

```
# Burbujas hasta el controlador predeterminado que genera el mensaje
withCallingHandlers(
  message = function(cnd) cat("Level 2\n"),
  withCallingHandlers(
    message = function(cnd) cat("Level 1\n"),
    message("Hello")
  )
)
#> Level 1
#> Level 2
#> Hello

# Burbujas en tryCatch
tryCatch(
  message = function(cnd) cat("Level 2\n"),
  withCallingHandlers(
    message = function(cnd) cat("Level 1\n"),
    message("Hello")
  )
)
#> Level 1
#> Level 2
```

Si desea evitar la condición "burbujeante" pero aún ejecuta el resto del código en el bloque, debe silenciarlo explícitamente con `rlang::cnd_muffle()`:

```
# Silencia el controlador predeterminado que imprime los mensajes.
withCallingHandlers(
  message = function(cnd) {
    cat("Level 2\n")
    cnd_muffle(cnd)
  },
```

```
  withCallingHandlers(
    message = function(cnd) cat("Level 1\n"),
    message("Hello")
  )
)
#> Level 1
#> Level 2

# Silencia el controlador de nivel 2 y el controlador por defecto
withCallingHandlers(
  message = function(cnd) cat("Level 2\n"),
  withCallingHandlers(
    message = function(cnd) {
      cat("Level 1\n")
      cnd_muffle(cnd)
    },
    message("Hello")
  )
)
#> Level 1
```

8.4.4. Pilas de llamadas

Para completar la sección, existen algunas diferencias importantes entre las pilas de llamadas de los controladores de salida y de llamada. Estas diferencias generalmente no son importantes, pero las incluyo aquí porque ocasionalmente las he encontrado útiles, ¡y no quiero olvidarme de ellas!

Es más fácil ver la diferencia configurando un pequeño ejemplo que usa `lobstr::cst()`:

```
f <- function() g()
g <- function() h()
h <- function() message("!")
```

Los controladores de llamadas se llaman en el contexto de la llamada que señaló la condición:

```
withCallingHandlers(f(), message = function(cnd) {
  lobstr::cst()
  cnd_muffle(cnd)
})
#>
#>   1.  base::withCallingHandlers(...)
#>   2.  global f()
#>   3.    global g()
#>   4.      global h()
#>   5.        base::message("!")
#>   6.          base::withRestarts(...)
#>   7.            base (local) withOneRestart(expr, restarts[[1L]])
#>   8.              base (local) doWithOneRestart(return(expr), restart)
#>   9.          base::signalCondition(cond)
#>  10.  global `<fn>`(`<smplMssg>`)
#>  11.    lobstr::cst()
```

Mientras que los controladores existentes se llaman en el contexto de la llamada a `tryCatch()`:

```
tryCatch(f(), message = function(cnd) lobstr::cst())
#>
#>  1.  base::tryCatch(f(), message = function(cnd) lobstr::cst())
#>  2.    base (local) tryCatchList(expr, classes, parentenv, handlers)
#>  3.      base (local) tryCatchOne(expr, names, parentenv, handlers[[1L]])
```

```
#>  4.         value[[3L]](cond)
#>  5.           lobstr::cst()
```

8.4.5. Ejercicios

1. ¿Qué información adicional contiene la condición generada por `abort()` en comparación con la condición generada por `stop()`, es decir, cuál es la diferencia entre estos dos objetos? Lea la ayuda de `?abort` para obtener más información.

```
catch_cnd(stop("An error"))
catch_cnd(abort("An error"))
```

2. Prediga los resultados de evaluar el siguiente código

```
show_condition <- function(code) {
  tryCatch(
    error = function(cnd) "error",
    warning = function(cnd) "warning",
    message = function(cnd) "message",
    {
      code
      NULL
    }
  )
}

show_condition(stop("!"))
show_condition(10)
show_condition(warning("?!"))
show_condition({
  10
  message("?")
```

```
  warning("?!")
})
```

3. Explique los resultados de ejecutar este código:

```
withCallingHandlers(
  message = function(cnd) message("b"),
  withCallingHandlers(
    message = function(cnd) message("a"),
    message("c")
  )
)
#> b
#> a
#> b
#> c
```

4. Lea el código fuente de `catch_cnd()` y explique cómo funciona.

5. ¿Cómo podría reescribir `show_condition()` para usar un solo controlador?

8.5. Condiciones personalizadas

Uno de los desafíos del manejo de errores en R es que la mayoría de las funciones generan una de las condiciones integradas, que contienen solo un "mensaje" y una "llamada". Eso significa que si desea detectar un tipo específico de error, solo puede trabajar con el texto del mensaje de error. Esto es propenso a errores, no solo porque el mensaje puede cambiar con el tiempo, sino también porque los mensajes se pueden traducir a otros idiomas.

Afortunadamente, R tiene una característica poderosa, pero poco utilizada: la capacidad de crear condiciones personalizadas que pueden contener

metadatos adicionales. Crear condiciones personalizadas es un poco complicado en base R, pero `rlang::abort()` lo hace muy fácil ya que puede proporcionar una `.subclass` personalizada y metadatos adicionales.

El siguiente ejemplo muestra el patrón básico. Recomiendo usar la siguiente estructura de llamadas para condiciones personalizadas. Esto aprovecha la coincidencia de argumentos flexibles de R para que el nombre del tipo de error aparezca primero, seguido del texto de cara al usuario, seguido de los metadatos personalizados.

```
abort(
  "error_not_found",
  message = "Path `blah.csv` not found",
  path = "blah.csv"
)
#> Error:
#> ! Path `blah.csv` not found
```

Las condiciones personalizadas funcionan igual que las condiciones normales cuando se usan de forma interactiva, pero permiten que los controladores hagan mucho más.

8.5.1. Motivación

To explore these ideas in more depth, let's take `base::log()`. It does the minimum when throwing errors caused by invalid arguments:

```
log(letters)
#> Error in log(letters): non-numeric argument to mathematical function
log(1:10, base = letters)
#> Error in log(1:10, base = letters): non-numeric argument to mathematical f
```

Creo que podemos hacerlo mejor siendo explícitos sobre qué argumento es el problema (es decir, `x` o `base`) y diciendo cuál es la entrada problemática (no solo cuál no es).

```
my_log <- function(x, base = exp(1)) {
  if (!is.numeric(x)) {
    abort(paste0(
      "`x` must be a numeric vector; not ", typeof(x), "."
    ))
  }
  if (!is.numeric(base)) {
    abort(paste0(
      "`base` must be a numeric vector; not ", typeof(base), "."
    ))
  }

  base::log(x, base = base)
}
```

Esto nos da:

```
my_log(letters)
#> Error in `my_log()`:
#> ! `x` must be a numeric vector; not character.
my_log(1:10, base = letters)
#> Error in `my_log()`:
#> ! `base` must be a numeric vector; not character.
```

Esta es una mejora para el uso interactivo, ya que es más probable que los mensajes de error guíen al usuario hacia una solución correcta. Sin embargo, no son mejores si desea manejar los errores mediante programación: todos los metadatos útiles sobre el error se atascan en una sola cadena.

8.5.2. Señalización

Construyamos alguna infraestructura para mejorar esta situación. Comenzaremos proporcionando una función `abort()` personalizada para argumentos incorrectos. Esto está un poco generalizado para el ejemplo en cuestión, pero refleja patrones comunes que he visto en otras funciones. El patrón es bastante simple. Creamos un buen mensaje de error para el usuario, usando `glue::glue()`, y almacenamos metadatos en la llamada de condición para el desarrollador.

```
abort_bad_argument <- function(arg, must, not = NULL) {
  msg <- glue::glue("`{arg}` must {must}")
  if (!is.null(not)) {
    not <- typeof(not)
    msg <- glue::glue("{msg}; not {not}.")
  }

  abort("error_bad_argument",
    message = msg,
    arg = arg,
    must = must,
    not = not
  )
}
```

Si desea generar un error personalizado sin agregar una dependencia en rlang, puede crear un objeto de condición "a mano" y luego pasarlo a `stop()`:

```
stop_custom <- function(.subclass, message, call = NULL, ...) {
  err <- structure(
    list(
      message = message,
      call = call,
      ...
    ),
    class = c(.subclass, "error", "condition")
  )
  stop(err)
}

err <- catch_cnd(
  stop_custom("error_new", "This is a custom error", x = 10)
)
class(err)
err$x
```

Ahora podemos reescribir `my_log()` para usar este nuevo ayudante:

```
my_log <- function(x, base = exp(1)) {
  if (!is.numeric(x)) {
    abort_bad_argument("x", must = "be numeric", not = x)
  }
  if (!is.numeric(base)) {
    abort_bad_argument("base", must = "be numeric", not = base)
  }

  base::log(x, base = base)
}
```

`my_log()` en sí mismo no es mucho más corto, pero es un poco más significativo y asegura que los mensajes de error para argumentos incorrectos sean consistentes en todas las funciones. Produce los mismos mensajes de error interactivos que antes:

```
my_log(letters)
#> Error in `abort_bad_argument()`:
#> ! `x` must be numeric; not character.
my_log(1:10, base = letters)
#> Error in `abort_bad_argument()`:
#> ! `base` must be numeric; not character.
```

8.5.3. Controlar

Estos objetos de condición estructurados son mucho más fáciles de programar. El primer lugar en el que podría querer usar esta capacidad es al probar su función. Las pruebas unitarias no son un tema de este libro (consulte los paquetes R para obtener más detalles), pero los conceptos básicos son fáciles de entender. El siguiente código captura el error y luego afirma que tiene la estructura que esperamos.

```
library(testthat)

err <- catch_cnd(my_log("a"))
expect_s3_class(err, "error_bad_argument")
expect_equal(err$arg, "x")
expect_equal(err$not, "character")
```

También podemos usar la clase (`error_bad_argument`) en `tryCatch()` para manejar solo ese error específico:

```
tryCatch(
  error_bad_argument = function(cnd) "bad_argument",
  error = function(cnd) "other error",
  my_log("a")
)
#> [1] "bad_argument"
```

Cuando se usa `tryCatch()` con múltiples controladores y clases personalizadas, se llama al primer controlador que coincida con cualquier clase en el vector de clase de la señal, no a la mejor coincidencia. Por este motivo, debe asegurarse de poner primero los controladores más específicos. El siguiente código no hace lo que cabría esperar:

```
tryCatch(
  error = function(cnd) "other error",
  error_bad_argument = function(cnd) "bad_argument",
  my_log("a")
)
#> [1] "other error"
```

8.5.4. Ejercicios

1. Dentro de un paquete, ocasionalmente es útil verificar que un paquete esté instalado antes de usarlo. Escriba una función que verifique si un paquete está instalado (con `requireNamespace("pkg", quietly = FALSE)`) y, si no, arroja una condición personalizada que incluye el nombre del paquete en los metadatos.

2. Dentro de un paquete, a menudo debe detenerse con un error cuando algo no está bien. Otros paquetes que dependen de su paquete pueden verse tentados a verificar estos errores en sus pruebas unitarias. ¿Cómo podría ayudar a estos paquetes a evitar confiar en el

mensaje de error que es parte de la interfaz de usuario en lugar de la API y que podría cambiar sin previo aviso?

8.6. Aplicaciones

Ahora que ha aprendido las herramientas básicas del sistema de condiciones de R, es hora de sumergirse en algunas aplicaciones. El objetivo de esta sección no es mostrar todos los usos posibles de `tryCatch()` y `withCallingHandlers()`, sino ilustrar algunos patrones comunes que surgen con frecuencia. Con suerte, esto hará que fluya su creatividad, de modo que cuando encuentre un nuevo problema, pueda encontrar una solución útil.

8.6.1. Valor de falla

Hay algunos patrones `tryCatch()` simples, pero útiles, basados en la devolución de un valor del controlador de errores. El caso más simple es un contenedor para devolver un valor predeterminado si ocurre un error:

```
fail_with <- function(expr, value = NULL) {
  tryCatch(
    error = function(cnd) value,
    expr
  )
}

fail_with(log(10), NA_real_)
#> [1] 2.3
fail_with(log("x"), NA_real_)
#> [1] NA
```

Una aplicación más sofisticada es `base::try()`. A continuación, `try2()` extrae la esencia de `base::try()`; la función real es más complicada para hacer que el mensaje de error se parezca más a lo que vería si no se usara `tryCatch()`.

```
try2 <- function(expr, silent = FALSE) {
  tryCatch(
    error = function(cnd) {
      msg <- conditionMessage(cnd)
      if (!silent) {
        message("Error: ", msg)
      }
      structure(msg, class = "try-error")
    },
    expr
  )
}

try2(1)
#> [1] 1
try2(stop("Hi"))
#> Error: Hi
#> [1] "Hi"
#> attr(,"class")
#> [1] "try-error"
try2(stop("Hi"), silent = TRUE)
#> [1] "Hi"
#> attr(,"class")
#> [1] "try-error"
```

8.6.2. Valores de éxito y fracaso.

Podemos ampliar este patrón para devolver un valor si el código se evalúa correctamente (`success_val`) y otro si falla (`error_val`). Este patrón solo requiere un pequeño truco: evaluar el código proporcionado por el usuario y luego `success_val`. Si el código arroja un error, nunca llegaremos a `success_val` y, en su lugar, devolveremos `error_val`.

```
foo <- function(expr) {
  tryCatch(
    error = function(cnd) error_val,
    {
      expr
      success_val
    }
  )
}
```

Podemos usar esto para determinar si una expresión falla:

```
does_error <- function(expr) {
  tryCatch(
    error = function(cnd) TRUE,
    {
      expr
      FALSE
    }
  )
}
```

O para capturar cualquier condición, como simplemente `rlang::catch_cnd()`:

```
catch_cnd <- function(expr) {
  tryCatch(
    condition = function(cnd) cnd,
    {
      expr
      NULL
    }
  )
}
```

También podemos usar este patrón para crear una variante `try()`. Un desafío con `try()` es que es un poco difícil determinar si el código tuvo éxito o falló. En lugar de devolver un objeto con una clase especial, creo que es un poco mejor devolver una lista con dos componentes `result` y `error`.

```
safety <- function(expr) {
  tryCatch(
    error = function(cnd) {
      list(result = NULL, error = cnd)
    },
    list(result = expr, error = NULL)
  )
}

str(safety(1 + 10))
#> List of 2
#>  $ result: num 11
#>  $ error : NULL
str(safety(stop("Error!")))
#> List of 2
#>  $ result: NULL
#>  $ error :List of 2
```

```
#>   ..$ message: chr "Error!"
#>   ..$ call   : language doTryCatch(return(expr), name, parentenv, h..
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

(Esto está estrechamente relacionado con `purrr::safely()`, un operador de función, al que volveremos en la Section 11.2.1.)

8.6.3. Renuncia

Además de devolver valores predeterminados cuando se señala una condición, los controladores se pueden usar para generar mensajes de error más informativos. Una aplicación simple es hacer una función que funcione como `options(warn = 2)` para un solo bloque de código. La idea es simple: manejamos las advertencias lanzando un error:

```
warning2error <- function(expr) {
  withCallingHandlers(
    warning = function(cnd) abort(conditionMessage(cnd)),
    expr
  )
}
```

```
warning2error({
  x <- 2 ^ 4
  warn("Hello")
})
#> Error:
#> ! Hello
```

Podrías escribir una función similar si estuvieras tratando de encontrar la fuente de un mensaje molesto. Más sobre esto en la Section 22.6.

8.6.4. Registro

Otro patrón común es registrar las condiciones para una investigación posterior. El nuevo desafío aquí es que los controladores de llamadas se llaman solo por sus efectos secundarios, por lo que no podemos devolver valores, sino que necesitamos modificar algún objeto en su lugar.

```
catch_cnds <- function(expr) {
  conds <- list()
  add_cond <- function(cnd) {
    conds <<- append(conds, list(cnd))
    cnd_muffle(cnd)
  }

  withCallingHandlers(
    message = add_cond,
    warning = add_cond,
    expr
  )

  conds
}

catch_cnds({
  inform("a")
  warn("b")
  inform("c")
})
#> [[1]]
#> <message/rlang_message>
#> Message:
#> a
#>
#> [[2]]
```

```
#> <warning/rlang_warning>
#> Warning:
#> b
#>
#> [[3]]
#> <message/rlang_message>
#> Message:
#> c
```

¿Qué sucede si también desea capturar errores? Deberá envolver el `withCallingHandlers()` en un `tryCatch()`. Si se produce un error, será la última condición.

```
catch_cnds <- function(expr) {
  conds <- list()
  add_cond <- function(cnd) {
    conds <<- append(conds, list(cnd))
    cnd_muffle(cnd)
  }

  tryCatch(
    error = function(cnd) {
      conds <<- append(conds, list(cnd))
    },
    withCallingHandlers(
      message = add_cond,
      warning = add_cond,
      expr
    )
  )

  conds
}
```

```
catch_cnds({
  inform("a")
  warn("b")
  abort("C")
})
#> [[1]]
#> <message/rlang_message>
#> Message:
#> a
#>
#> [[2]]
#> <warning/rlang_warning>
#> Warning:
#> b
#>
#> [[3]]
#> <error/rlang_error>
#> Error:
#> ! C
#> ---
#> Backtrace:
#>
```

Esta es la idea clave que subyace al paquete de evaluación (Wickham and Xie 2018) que impulsa a knitr: captura cada salida en una estructura de datos especial para que pueda reproducirse más tarde. En general, el paquete de evaluación es mucho más complicado que el código aquí porque también necesita manejar gráficos y salida de texto.

8.6.5. Sin comportamiento predeterminado

Un último patrón útil es señalar una condición que no hereda de `message`, `warning` o `error`. Debido a que no hay un comportamiento predeterminado, esto significa que la condición no tiene efecto a menos que el usuario lo solicite específicamente. Por ejemplo, podría imaginar un sistema de registro basado en condiciones:

```
log <- function(message, level = c("info", "error", "fatal")) {
  level <- match.arg(level)
  signal(message, "log", level = level)
}
```

Cuando llamas a `log()`, se señala una condición, pero no sucede nada porque no tiene un controlador predeterminado:

```
log("This code was run")
```

Para activar el registro, necesita un controlador que haga algo con la condición `log`. A continuación defino una función `record_log()` que registrará todos los mensajes de registro en un archivo:

```
record_log <- function(expr, path = stdout()) {
  withCallingHandlers(
    log = function(cnd) {
      cat(
        "[", cnd$level, "] ", cnd$message, "\n", sep = "",
        file = path, append = TRUE
      )
    },
    expr
  )
}
```

```
record_log(log("Hello"))
#> [info] Hello
```

Incluso podría imaginar la superposición con otra función que le permita suprimir selectivamente algunos niveles de registro.

```
ignore_log_levels <- function(expr, levels) {
  withCallingHandlers(
    log = function(cnd) {
      if (cnd$level %in% levels) {
        cnd_muffle(cnd)
      }
    },
    expr
  )
}

record_log(ignore_log_levels(log("Hello"), "info"))
```

Si crea un objeto de condición a mano y lo señala con `signalCondition()`, `cnd_muffle()` no funcionará. En su lugar, debe llamarlo con un reinicio de mufla definido, así:

```
withRestarts(signalCondition(cond), muffle = function() NULL)
```

Los reinicios están actualmente fuera del alcance del libro, pero sospecho que se incluirán en la tercera edición.

8.6.6. Ejercicios

1. Cree `suppressConditions()` que funcione como `suppressMessages()` y `suppressWarnings()` pero suprima todo. Piense cuidadosamente

acerca de cómo debe manejar los errores.

2. Compare las siguientes dos implementaciones de `message2error()`. ¿Cuál es la principal ventaja de `withCallingHandlers()` en este escenario? (Sugerencia: mire cuidadosamente el rastreo).

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

3. ¿Cómo modificaría la definición de `catch_cnds()` si quisiera recrear la combinación original de advertencias y mensajes?

4. ¿Por qué es peligroso atrapar interrupciones? Ejecute este código para averiguarlo.

```
bottles_of_beer <- function(i = 99) {
  message(
    "There are ", i, " bottles of beer on the wall, ",
    i, " bottles of beer."
  )
  while(i > 0) {
    tryCatch(
      Sys.sleep(1),
      interrupt = function(err) {
        i <<- i - 1
        if (i > 0) {
          message(
            "Take one down, pass it around, ", i,
            " bottle", if (i > 1) "s", " of beer on the wall."
          )
        }
      }
```

```
    )
  }
  message(
    "No more bottles of beer on the wall, ",
    "no more bottles of beer."
  )
}
```

8.7. Respuestas de la prueba

1. `error`, `warning`, y `message`.
2. Podrías usar `try()` o `tryCatch()`.
3. `tryCatch()` crea controladores existentes que finalizarán la ejecución del código envuelto; `withCallingHandlers()` crea controladores de llamadas que no afectan la ejecución del código envuelto.
4. Porque luego puede capturar tipos específicos de error con `tryCatch()`, en lugar de depender de la comparación de cadenas de errores, lo cual es arriesgado, especialmente cuando se traducen los mensajes.

Part II.

Programación funcional

Introducción

R, en esencia, es un lenguaje **funcional**. Esto significa que tiene ciertas propiedades técnicas, pero lo más importante es que se presta a un estilo de resolución de problemas centrado en funciones. A continuación, daré una breve descripción de la definición técnica de un *lenguaje* funcional, pero en este libro me centraré principalmente en el *estilo* funcional de programación, porque creo que se adapta muy bien a los tipos de problemas comúnmente se encuentra al hacer análisis de datos.

Recientemente, las técnicas funcionales han experimentado un gran interés porque pueden producir soluciones eficientes y elegantes para muchos problemas modernos. Un estilo funcional tiende a crear funciones que pueden analizarse fácilmente de forma aislada (es decir, utilizando solo información local) y, por lo tanto, a menudo es mucho más fácil de optimizar o paralelizar automáticamente. Las debilidades tradicionales de los lenguajes funcionales, el rendimiento más bajo y, a veces, el uso impredecible de la memoria, se han reducido mucho en los últimos años. La programación funcional es complementaria a la programación orientada a objetos, que ha sido el paradigma de programación dominante durante las últimas décadas.

Lenguajes de programacion funcional

Cada lenguaje de programación tiene funciones, entonces, ¿qué hace que un lenguaje de programación sea funcional? Hay muchas definiciones de lo que hace que un lenguaje sea funcional, pero hay dos hilos comunes.

En primer lugar, los lenguajes funcionales tienen **funciones de primera clase**, funciones que se comportan como cualquier otra estructura de datos. En R, esto significa que puede hacer muchas de las cosas con una función que puede hacer con un vector: puede asignarlas a variables, almacenarlas en listas, pasarlas como argumentos a otras funciones, crearlas dentro de funciones y incluso devolverlos como resultado de una función.

En segundo lugar, muchos lenguajes funcionales requieren que las funciones sean **puras**. Una función es pura si cumple dos propiedades:

- La salida solo depende de las entradas, es decir, si lo vuelve a llamar con las mismas entradas, obtendrá las mismas salidas. Esto excluye funciones como `runif()`, `read.csv()` o `Sys.time()` que pueden devolver valores diferentes.

- La función no tiene efectos secundarios, como cambiar el valor de una variable global, escribir en el disco o mostrar en la pantalla. Esto excluye funciones como `print()`, `write.csv()` y `<-`.

Las funciones puras son mucho más fáciles de razonar, pero obviamente tienen desventajas significativas: imagine hacer un análisis de datos en el que no pueda generar números aleatorios o leer archivos del disco.

Estrictamente hablando, R no es un *lenguaje* de programación funcional porque no requiere que escribas funciones puras. Sin embargo, ciertamente puede adoptar un estilo funcional en partes de su código: no *tiene* que escribir funciones puras, pero a menudo *debería*. En mi experiencia, dividir el código en funciones que son extremadamente puras o extremadamente impuras tiende a generar un código que es más fácil de entender y se extiende a nuevas situaciones.

Estilo funcional

Es difícil describir exactamente qué es un *estilo* funcional, pero en general creo que significa descomponer un gran problema en partes más pequeñas y luego resolver cada parte con una función o combinación de funciones. Cuando usa un estilo funcional, se esfuerza por descomponer los componentes del problema en funciones aisladas que operan de forma independiente. Cada función tomada por sí sola es simple y fácil de entender; la complejidad se maneja componiendo funciones de varias maneras.

Los siguientes tres capítulos analizan las tres técnicas funcionales clave que lo ayudan a descomponer los problemas en partes más pequeñas:

- El Chapter 9 muestra cómo reemplazar muchos bucles for con **funcionales** que son funciones (como `lapply()`) que toman otra función como argumento. Los funcionales le permiten tomar una función que resuelve el problema para una sola entrada y generalizarla para manejar cualquier número de entradas. Los funcionales son, con mucho, la técnica más importante y los usará todo el tiempo en el análisis de datos.

- El Chapter 10 introduce **fábricas de funciones**: funciones que crean funciones. Las fábricas de funciones se usan con menos frecuencia que las funcionales, pero pueden permitirle dividir elegantemente el trabajo entre diferentes partes de su código.

- El Chapter 11 le muestra cómo crear **operadores de función**: funciones que toman funciones como entrada y producen funciones como salida. Son como los adverbios, porque normalmente modifican el funcionamiento de una función.

En conjunto, estos tipos de funciones se denominan **funciones de orden superior** y completan una tabla de dos por dos:

In \ Out	Vector	Function
Vector	Regular function	Function factory
Function	Functional	Function operator

9. Funcionales

9.1. Introducción

> Para volverse significativamente más confiable, el código debe volverse más transparente. En particular, las condiciones anidadas y los bucles deben verse con gran sospecha. Los flujos de control complicados confunden a los programadores. El código desordenado a menudo esconde errores.
>
> — Bjarne Stroustrup

Un **funcional** es una función que toma una función como entrada y devuelve un vector como salida. Aquí hay un funcional simple: llama a la función proporcionada como entrada con 1000 números uniformes aleatorios.

```
randomise <- function(f) f(runif(1e3))
randomise(mean)
#> [1] 0.506
randomise(mean)
#> [1] 0.501
randomise(sum)
#> [1] 489
```

Lo más probable es que ya hayas usado un funcional. Es posible que haya utilizado reemplazos de bucle for como `lapply()`, `apply()` y `tapply()` de

base R; o `map()` de purrr; o quizás hayas usado un funcional matemático como `integrate()` u `optim()`.

Un uso común de los funcionales es como una alternativa a los bucles for. Los bucles for tienen mala reputación en R porque mucha gente cree que son lentos[^funcionals-1], pero la verdadera desventaja de los bucles for es que son muy flexibles: un bucle transmite que estás iterando, pero no lo que deberías terminar con los resultados. Así como es mejor usar `while` que `repeat`, y es mejor usar `for` que `while` (Section 5.3.2), es mejor usar un funcional que `for`. Cada funcional está diseñado para una tarea específica, por lo que cuando reconoce el funcional, inmediatamente sabe por qué se está utilizando.

Si es un usuario experimentado de bucles for, cambiar a funcionales suele ser un ejercicio de coincidencia de patrones. Miras el bucle for y encuentras un funcional que coincida con la forma básica. Si no existe uno, no intente torturar un funcional existente para que se ajuste a la forma que necesita. ¡En su lugar, déjalo como un bucle for! (O una vez que haya repetido el mismo bucle dos o más veces, tal vez piense en escribir su propio funcional).

Outline

- La Section 9.2 introduce tu primer funcional: `purrr :: map ()`.
- La Section 9.3 demuestra cómo puede combinar múltiples funciones simples para resolver un problema más complejo y analiza cómo el estilo purrr difiere de otros enfoques.
- La Section 9.4 te enseña alrededor de 18 (!!) variantes importantes de `purrr::map()`. Afortunadamente, su diseño ortogonal los hace fáciles de aprender, recordar y dominar.

- La Section 9.5 introduce un nuevo estilo de funcional: `purrr::reduce()`. `reduce()` reduce sistemáticamente un vector a un solo resultado mediante la aplicación de una función que toma dos entradas.
- La Section 9.6 te enseña acerca de los predicados: funciones que devuelven un solo `TRUE` o `FALSE`, y la familia de funciones que los usan para resolver problemas comunes.
- La Section 9.7 revisa algunos funcionales en base R que no son miembros de las familias map, reduce o predicate.

Requsisitos previos

Este capítulo se centrará en las funciones proporcionadas por el paquete purrr (Henry and Wickham 2018a). Estas funciones tienen una interfaz consistente que facilita la comprensión de las ideas clave que sus equivalentes básicos, que han crecido orgánicamente durante muchos años. Compararé y contrastaré las funciones básicas de R a medida que avanzamos, y luego terminaré el capítulo con una discusión de las funciones básicas que no tienen equivalentes purrr.

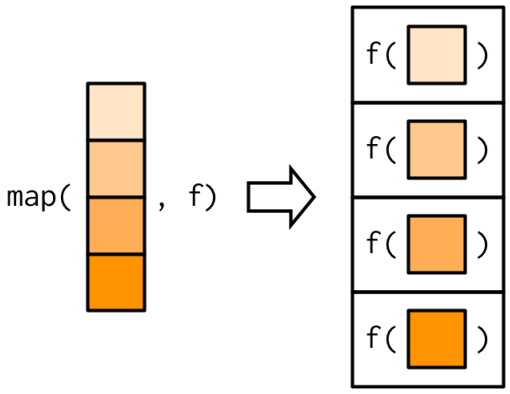
```
library(purrr)
```

9.2. My first functional: `map()`

El funcional más fundamental es `purrr::map()`[^funcionals-2]. Toma un vector y una función, llama a la función una vez por cada elemento del vector y devuelve los resultados en una lista. En otras palabras, `map(1:3, f)` es equivalente a `list(f(1), f(2), f(3))`.

```
triple <- function(x) x * 3
map(1:3, triple)
#> [[1]]
#> [1] 3
#>
#> [[2]]
#> [1] 6
#>
#> [[3]]
#> [1] 9
```

O, gráficamente:



Quizás se pregunte por qué esta función se llama `map()`. ¿Qué tiene que ver con representar las características físicas de la tierra o el mar 🗺? De hecho, el significado proviene de las matemáticas donde *mapa* se refiere a "una operación que asocia cada elemento de un conjunto dado con uno o más elementos de un segundo conjunto". Esto tiene sentido aquí porque `map()` define un mapeo de un vector a otro. (*"Map"* también tiene la agradable

propiedad de ser corto, lo cual es útil para un bloque de construcción tan fundamental).

La implementación de `map()` es bastante simple. Asignamos una lista de la misma longitud que la entrada y luego completamos la lista con un bucle for. El corazón de la implementación es solo un puñado de líneas de código:

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

La verdadera función `purrr::map()` tiene algunas diferencias: está escrita en C para aprovechar hasta el último ápice de rendimiento, conserva los nombres y admite algunos atajos que aprenderá en la Section 9.2.2.

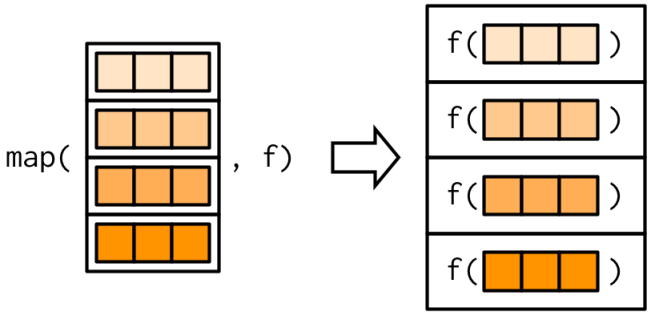
El equivalente básico de `map()` es `lapply()`. La única diferencia es que `lapply()` no es compatible con los asistentes que aprenderá a continuación, por lo que si solo está usando `map()` de purrr, puede omitir la dependencia adicional y usar `lapply()` directamente.

9.2.1. Producción de vectores atómicos

`map()` devuelve una lista, lo que la convierte en la más general de la familia de mapas porque puedes poner cualquier cosa en una lista. Pero es un inconveniente devolver una lista cuando lo haría una estructura de datos más simple, por lo que hay cuatro variantes más específicas: `map_lgl()`, `map_int()`, `map_dbl()` y `map_chr()`. Cada uno devuelve un vector atómico del tipo especificado:

```
# map_chr() siempre devuelve un vector de caracteres
map_chr(mtcars, typeof)
#>      mpg      cyl     disp       hp     drat       wt     qsec
#> "double" "double" "double" "double" "double" "double" "double"
#>       vs       am     gear     carb
#> "double" "double" "double" "double"

# map_lgl() siempre devuelve un vector de valore boleanos
map_lgl(mtcars, is.double)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

# map_int() siempre devuelve un vector de números enteros
n_unique <- function(x) length(unique(x))
map_int(mtcars, n_unique)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6

# map_dbl() siempre devuelve un vector doble
map_dbl(mtcars, mean)
#>     mpg     cyl    disp      hp    drat      wt    qsec      vs
#>  20.091   6.188 230.722 146.688   3.597   3.217  17.849   0.438
#>      am    gear    carb
#>   0.406   3.688   2.812
```

purrr usa la convención de que los sufijos, como `_dbl()`, se refieren a la salida. Todas las funciones `map_*()` pueden tomar cualquier tipo de vector como entrada. Estos ejemplos se basan en dos hechos: `mtcars` es un data frame y los data frames son listas que contienen vectores de la misma longitud. Esto es más obvio si dibujamos un data frame con la misma orientación que el vector:



Todas las funciones de mapa siempre devuelven un vector de salida de la misma longitud que la entrada, lo que implica que cada llamada a `.f` debe devolver un solo valor. Si no es así, obtendrá un error:

```
pair <- function(x) c(x, x)
map_dbl(1:2, pair)
#> Error in `map_dbl()`:
#>   In index: 1.
#> Caused by error:
#> ! Result must be length 1, not 2.
```

Esto es similar al error que obtendrá si `.f` devuelve el tipo de resultado incorrecto:

```
map_dbl(1:2, as.character)
#> Error in `map_dbl()`:
#>   In index: 1.
#> Caused by error:
#> ! Can't coerce from a string to a double.
```

En cualquier caso, a menudo es útil volver a `map()`, porque `map()` puede aceptar cualquier tipo de salida. Eso le permite ver la salida problemática y averiguar qué hacer con ella.

```
map(1:2, pair)
#> [[1]]
#> [1] 1 1
#>
#> [[2]]
#> [1] 2 2
map(1:2, as.character)
#> [[1]]
#> [1] "1"
#>
#> [[2]]
#> [1] "2"
```

Base R tiene dos funciones de aplicación que pueden devolver vectores atómicos: `sapply()` y `vapply()`. Te recomiendo que evites `sapply()` porque intenta simplificar el resultado, por lo que puede devolver una lista, un vector o una matriz. Esto dificulta la programación y debe evitarse en entornos no interactivos. `vapply()` es más seguro porque le permite proporcionar una plantilla, `FUN.VALUE`, que describe la forma de salida. Si no quieres usar purrr, te recomiendo que siempre uses `vapply()` en tus funciones, no `sapply()`. La principal desventaja de `vapply()` es su verbosidad: por ejemplo, el equivalente a `map_dbl(x, mean, na.rm = TRUE)` es `vapply(x, mean, na.rm = TRUE, FUN.VALUE = doble(1))`.

9.2.2. Funciones y accesos directos anónimos

En lugar de usar `map()` con una función existente, puede crear una función anónima en línea (como se menciona en la Section 6.2.3):

```
map_dbl(mtcars, function(x) length(unique(x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

Las funciones anónimas son muy útiles, pero la sintaxis es detallada. Así que purrr admite un atajo especial:

```
map_dbl(mtcars, ~ length(unique(.x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

Esto funciona porque todas las funciones purrr traducen fórmulas, creadas por `~` (pronunciado "twiddle"), en funciones. Puedes ver lo que sucede detrás de escena llamando a `as_mapper()`:

```
as_mapper(~ length(unique(.x)))
#> <lambda>
#> function (..., .x = ..1, .y = ..2, . = ..1)
#> length(unique(.x))
#> attr(,"class")
#> [1] "rlang_lambda_function" "function"
```

Los argumentos de la función parecen un poco extravagantes pero le permiten referirse a `.` para funciones de un argumento, `.x` y `.y` para funciones de dos argumentos, y `..1`, `..2`, `. .3`, etc., para funciones con un número arbitrario de argumentos. `.` permanece para la compatibilidad con versiones anteriores, pero no recomiendo usarlo porque se confunde fácilmente con el `.` utilizado por la canalización de magrittr.

Este atajo es particularmente útil para generar datos aleatorios:

```
x <- map(1:3, ~ runif(2))
str(x)
#> List of 3
#>  $ : num [1:2] 0.281 0.53
#>  $ : num [1:2] 0.433 0.917
#>  $ : num [1:2] 0.0275 0.8249
```

Reserve esta sintaxis para funciones cortas y simples. Una buena regla general es que si su función abarca líneas o usa `{}`, es hora de darle un nombre.

Las funciones del mapa también tienen atajos para extraer elementos de un vector, impulsados por `purrr::pluck()`. Puede utilizar un vector de caracteres para seleccionar elementos por nombre, un vector entero para seleccionar por posición o una lista para seleccionar tanto por nombre como por posición. Estos son muy útiles para trabajar con listas profundamente anidadas, que a menudo surgen cuando se trabaja con JSON.

```
x <- list(
  list(-1, x = 1, y = c(2), z = "a"),
  list(-2, x = 4, y = c(5, 6), z = "b"),
  list(-3, x = 8, y = c(9, 10, 11))
)

# Selecciona por nombre
map_dbl(x, "x")
#> [1] 1 4 8

# O por posición
map_dbl(x, 1)
#> [1] -1 -2 -3

# Or por ambos
map_dbl(x, list("y", 1))
#> [1] 2 5 9

# Obtendrá un error si un componente no existe:
map_chr(x, "z")
#> Error in `map_chr()`:
#>   In index: 3.
#> Caused by error:
```

```
#> ! Result must be length 1, not 0.

# A menos que proporcione un valor .default
map_chr(x, "z", .default = NA)
#> [1] "a" "b" NA
```

En las funciones básicas de R, como `lapply()`, puede proporcionar el nombre de la función como una cadena. Esto no es tremendamente útil ya que `lapply(x, "f")` es casi siempre equivalente a `lapply(x, f)` y es más tipeo.

9.2.3. Pasar argumentos con ...

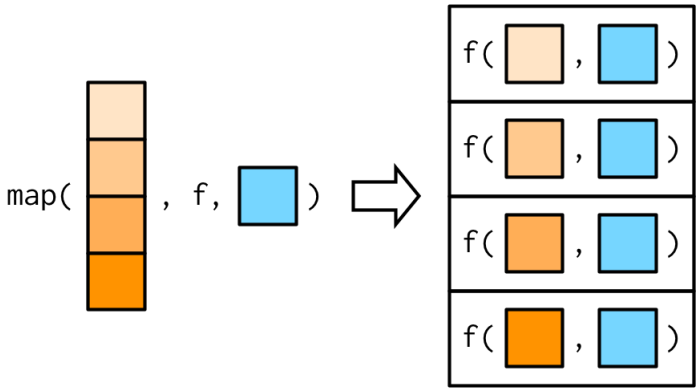
A menudo es conveniente pasar argumentos adicionales a la función que está llamando. Por ejemplo, es posible que desee pasar `na.rm = TRUE` junto con `mean()`. Una forma de hacerlo es con una función anónima:

```
x <- list(1:5, c(1:10, NA))
map_dbl(x, ~ mean(.x, na.rm = TRUE))
#> [1] 3.0 5.5
```

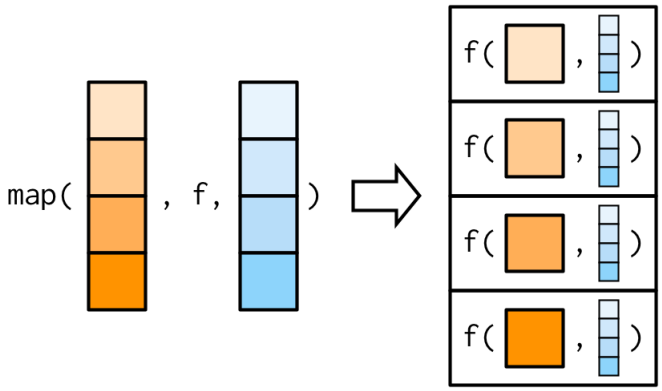
Pero debido a que las funciones del mapa pasan `...`, hay una forma más simple disponible:

```
map_dbl(x, mean, na.rm = TRUE)
#> [1] 3.0 5.5
```

Esto es más fácil de entender con una imagen: cualquier argumento que viene después de `f` en la llamada a `map()` se inserta *después* de los datos en llamadas individuales a `f()`:



Es importante tener en cuenta que estos argumentos no están descompuestos; o dicho de otra manera, `map()` solo se vectoriza sobre su primer argumento. Si un argumento después de `f` es un vector, se pasará como está:



(Aprenderá acerca de las variantes de mapa que *están* vectorizadas sobre múltiples argumentos en las Secciones Section 9.4.2 y Section 9.4.5.)

Tenga en cuenta que hay una sutil diferencia entre colocar argumentos adicionales dentro de una función anónima en comparación con pasarlos a `map()`. Ponerlos en una función anónima significa que serán evaluados cada vez que se ejecute `f()`, no solo una vez cuando llames a `map()`. Esto es más fácil de ver si hacemos que el argumento adicional sea aleatorio:

```
plus <- function(x, y) x + y

x <- c(0, 0, 0, 0)
map_dbl(x, plus, runif(1))
#> [1] 0.0625 0.0625 0.0625 0.0625
map_dbl(x, ~ plus(.x, runif(1)))
#> [1] 0.903 0.132 0.629 0.945
```

9.2.4. Nombres de argumentos

En los diagramas, he omitido los nombres de los argumentos para centrarme en la estructura general. Pero recomiendo escribir los nombres completos en su código, ya que lo hace más fácil de leer. `map(x, mean, 0.1)` es un código perfectamente válido, pero llamará `mean(x[[1]], 0.1)` por lo que depende de que el lector recuerde que el segundo argumento de `mean()` es `trim`. Para evitar una carga innecesaria en el cerebro del lector[^funcionals-3], sea amable y escriba `map(x, mean, trim = 0.1)`.

Esta es la razón por la que los argumentos de `map()` son un poco extraños: en lugar de ser `x` y `f`, son `.x` y `.f`. Es más fácil ver el problema que conduce a estos nombres usando `simple_map()` definido anteriormente. `simple_map()` tiene argumentos `x` y `f`, por lo que tendrá problemas cada vez que la función a la que llama tenga argumentos `x` o `f`:

```
bootstrap_summary <- function(x, f) {
  f(sample(x, replace = TRUE))
}
```

```
simple_map(mtcars, bootstrap_summary, f = mean)
#> Error in mean.default(x[[i]], ...): 'trim' must be numeric of length one
```

El error es un poco desconcertante hasta que recuerdas que la llamada a `simple_map()` es equivalente a `simple_map(x = mtcars, f = mean, bootstrap_summary)` porque la coincidencia con nombre supera a la coincidencia posicional.

Las funciones purrr reducen la probabilidad de que se produzca un conflicto de este tipo mediante el uso de `.f` y `.x` en lugar de las más comunes `f` y `x`. Por supuesto, esta técnica no es perfecta (porque la función a la que está llamando aún puede usar `.f` y `.x`), pero evita el 99% de los problemas. El 1% restante del tiempo, utilice una función anónima.

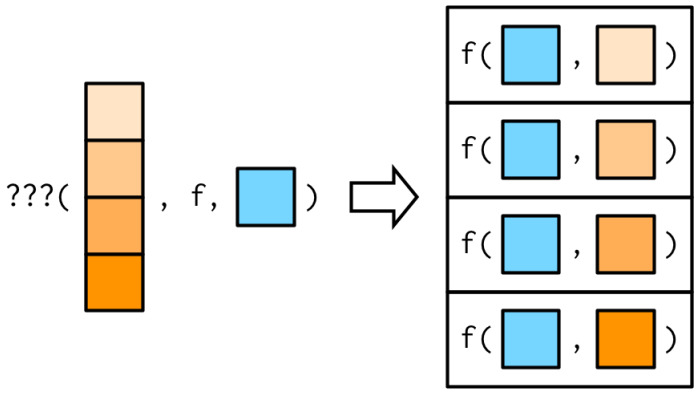
Las funciones base que transmiten `...` usan una variedad de convenciones de nomenclatura para evitar la coincidencia de argumentos no deseados:

- La familia apply utiliza principalmente letras mayúsculas (por ejemplo, `X` y `FUN`).
- `transform()` usa el prefijo más exótico `_`: esto hace que el nombre no sea sintáctico, por lo que siempre debe estar entre `` ` ``, como se describe en la Section 2.2.1. Esto hace que las coincidencias no deseadas sean extremadamente improbables.
- Otras funciones como `uniroot()` y `optim()` no hacen ningún esfuerzo por evitar conflictos, pero tienden a usarse con funciones especialmente creadas, por lo que es menos probable que se produzcan conflictos.

9.2.5. Variando otro argumento

Hasta ahora, el primer argumento de `map()` siempre se ha convertido en el primer argumento de la función. Pero, ¿qué sucede si el primer argumento

debe ser constante y desea variar un argumento diferente? ¿Cómo se obtiene el resultado en esta imagen?



Resulta que no hay forma de hacerlo directamente, pero hay dos trucos que puedes usar en su lugar. Para ilustrarlos, imagine que tengo un vector que contiene algunos valores inusuales y quiero explorar el efecto de diferentes cantidades de recorte al calcular la media. En este caso, el primer argumento de `mean()` será constante, y quiero variar el segundo argumento, `trim`.

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(1000)
```

- La técnica más simple es usar una función anónima para reorganizar el orden de los argumentos:

  ```
  map_dbl(trims, ~ mean(x, trim = .x))
  #> [1] -0.3500  0.0434  0.0354  0.0502
  ```

  Esto todavía es un poco confuso porque estoy usando `x` y `.x`. Puedes hacerlo un poco más claro abandonando el ayudante `~`:

```
map_dbl(trims, function(trim) mean(x, trim = trim))
#> [1] -0.3500  0.0434  0.0354  0.0502
```

- A veces, si quiere ser (demasiado) inteligente, puede aprovechar las reglas flexibles de coincidencia de argumentos de R (como se describe en la Section 6.8.2). Por ejemplo, en este ejemplo puede reescribir `mean(x, trim = 0.1)` como `mean(0.1, x = x)`, por lo que podría escribir la llamada a `map_dbl()` como:

  ```
  map_dbl(trims, mean, x = x)
  #> [1] -0.3500  0.0434  0.0354  0.0502
  ```

  No recomiendo esta técnica ya que se basa en la familiaridad del lector con el orden de los argumentos en `.f` y las reglas de coincidencia de argumentos de R.

Verá una alternativa más en la Section 9.4.5.

9.2.6. Ejercicios

1. Utilice `as_mapper()` para explorar cómo purrr genera funciones anónimas para los ayudantes de enteros, caracteres y listas. ¿Qué ayudante te permite extraer atributos? Lea la documentación para averiguarlo.

2. `map(1:3, ~ runif(2))` es un patrón útil para generar números aleatorios, pero `map(1:3, runif(2))` no lo es. ¿Por qué no? ¿Puede explicar por qué devuelve el resultado que lo hace?

3. Use la función `map()` apropiada para:

   a) Calcule la desviación estándar de cada columna en un data frame numéricos.

   b) Calcule la desviación estándar de cada columna numérica en un data frame mixto. (Sugerencia: deberá hacerlo en dos pasos).

c) Calcule el número de niveles para cada factor en un data frame.

4. El siguiente código simula el rendimiento de una prueba t para datos no normales. Extraiga el valor p de cada prueba, luego visualice.

```
trials <- map(1:100, ~ t.test(rpois(10, 10), rpois(7, 10)))
```

5. El siguiente código usa un mapa anidado dentro de otro mapa para aplicar una función a cada elemento de una lista anidada. ¿Por qué falla y qué debe hacer para que funcione?

```
x <- list(
  list(1, c(3, 9)),
  list(c(3, 6), 7, c(4, 7, 6))
)

triple <- function(x) x * 3
map(x, map, .f = triple)
#> Error in `map()`:
#>   In index: 1.
#> Caused by error in `.f()`:
#> ! unused argument (function (.x, .f, ..., .progress = FALSE)
#> {
#>     map_("list", .x, .f, ..., .progress = .progress)
#> })
```

6. Use `map()` para ajustar modelos lineales al conjunto de datos `mtcars` usando las fórmulas almacenadas en esta lista:

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

7. Ajuste el modelo `mpg ~ disp` a cada una de las réplicas de arranque de `mtcars` en la lista a continuación, luego extraiga el R^2 del ajuste del modelo (Sugerencia: puede calcular el R^2 con `summary()`.)

```
bootstrap <- function(df) {
  df[sample(nrow(df), replace = TRUE), , drop = FALSE]
}

bootstraps <- map(1:10, ~ bootstrap(mtcars))
```

9.3. Estilo Purrr

Antes de continuar explorando más variantes de mapas, echemos un vistazo rápido a cómo tiende a usar varias funciones purrr para resolver un problema moderadamente realista: ajustar un modelo a cada subgrupo y extraer un coeficiente del modelo. Para este ejemplo de juguete, voy a dividir el conjunto de datos `mtcars` en grupos definidos por el número de cilindros, utilizando la función base `split`:

```
by_cyl <- split(mtcars, mtcars$cyl)
```

Esto crea una lista de tres data frames: los automóviles con 4, 6 y 8 cilindros respectivamente.

Ahora imagine que queremos ajustar un modelo lineal, luego extraiga el segundo coeficiente (es decir, la pendiente). El siguiente código muestra cómo puede hacer eso con purrr:

```
by_cyl |>
  map(~ lm(mpg ~ wt, data = .x)) |>
  map(coef) |>
  map_dbl(2)
```

```
#>     4     6     8
#> -5.65 -2.78 -2.19
```

(Si no ha visto `|>`, la canalización, antes, se describe en la Section 6.3.)

Creo que este código es fácil de leer porque cada línea encapsula un solo paso, puedes distinguir fácilmente lo funcional de lo que hace, y los ayudantes purrr nos permiten describir de manera muy concisa qué hacer en cada paso.

¿Cómo atacarías este problema con la base R? Ciertamente *podría* reemplazar cada función purrr con la función base equivalente:

```
by_cyl |>
  lapply(function(data) lm(mpg ~ wt, data = data)) |>
  lapply(coef) |>
  vapply(function(x) x[[2]], double(1))
#>     4     6     8
#> -5.65 -2.78 -2.19
```

O, por supuesto, podría usar un bucle for:

```
slopes <- double(length(by_cyl))
for (i in seq_along(by_cyl)) {
  model <- lm(mpg ~ wt, data = by_cyl[[i]])
  slopes[[i]] <- coef(model)[[2]]
}
slopes
#> [1] -5.65 -2.78 -2.19
```

Es interesante notar que a medida que pasa de purrr a aplicar funciones básicas a bucles for, tiende a hacer más y más en cada iteración. En purrr iteramos 3 veces (`map()`, `map()`, `map_dbl()`), con funciones apply

iteramos dos veces (`lapply()`, `vapply()`), y con un for loop iteramos una vez. Prefiero más pasos, pero más simples, porque creo que hace que el código sea más fácil de entender y luego modificar.

9.4. Variantes de map

Hay 23 variantes principales de `map()`. Hasta ahora, ha aprendido acerca de cinco (`map()`, `map_lgl()`, `map_int()`, `map_dbl()` y `map_chr()`). Eso significa que tienes 18 (!!) más para aprender. Eso parece mucho, pero afortunadamente el diseño de purrr significa que solo necesitas aprender cinco nuevas ideas:

- Salida del mismo tipo que la entrada con `modify()`
- Iterar sobre dos entradas con `map2()`.
- Iterar con un índice usando `imap()`
- No devuelve nada con `walk()`.
- Iterar sobre cualquier número de entradas con `pmap()`.

La familia de funciones del mapa tiene entradas y salidas ortogonales, lo que significa que podemos organizar toda la familia en una matriz, con entradas en las filas y salidas en las columnas. Una vez que haya dominado la idea en una fila, puede combinarla con cualquier columna; una vez que haya dominado la idea en una columna, puede combinarla con cualquier fila. Esa relación se resume en el siguiente cuadro:

	List	Atómico	El mismo tipo	Nada
Un argumento	`map()`	`map_lgl()`, …	`modify()`	`walk()`
Dos argumentos	`map2()`	`map2_lgl()`, …	`modify2()`	`walk2()`
Un argumento + índice	`imap()`	`imap_lgl()`, …	`imodify()`	`iwalk()`

	List	Atómico	El mismo tipo	Nada
N argumentos	`pmap()`	`pmap_lgl()`, …	—	`pwalk()`

9.4.1. Mismo tipo de salida que de entrada: `modify()`

Imagina que quisieras duplicar cada columna en un data frame. Primero puede intentar usar `map()`, pero `map()` siempre devuelve una lista:

```
df <- data.frame(
  x = 1:3,
  y = 6:4
)

map(df, ~ .x * 2)
#> $x
#> [1] 2 4 6
#>
#> $y
#> [1] 12 10  8
```

Si desea mantener la salida como un data frame, puede usar `modify()`, que siempre devuelve el mismo tipo de salida que la entrada:

```
modify(df, ~ .x * 2)
#>   x  y
#> 1 2 12
#> 2 4 10
#> 3 6  8
```

A pesar del nombre, `modify()` no modifica en su lugar, devuelve una copia modificada, por lo que si desea modificar permanentemente `df`, debe asignarlo:

```
df <- modify(df, ~ .x * 2)
```

Como de costumbre, la implementación básica de `modify()` es simple y, de hecho, es incluso más simple que `map()` porque no necesitamos crear un nuevo vector de salida; podemos simplemente reemplazar progresivamente la entrada. (El código real es un poco complejo para manejar casos extremos con más gracia).

```
simple_modify <- function(x, f, ...) {
  for (i in seq_along(x)) {
    x[[i]] <- f(x[[i]], ...)
  }
  x
}
```

En la Section 9.6.2 aprenderá sobre una variante muy útil de `modify()`, llamada `modify_if()`. Esto le permite (p. ej.) solo duplicar columnas *numéricas* de un data frame con `modify_if(df, is.numeric, ~ .x * 2)`.

9.4.2. Dos entradas: `map2()` y amigos

`map()` se vectoriza sobre un único argumento, `.x`. Esto significa que solo varía `.x` cuando se llama a `.f`, y todos los demás argumentos se pasan sin cambios, por lo que no es adecuado para algunos problemas. Por ejemplo, ¿cómo encontraría una media ponderada cuando tiene una lista de observaciones y una lista de pesos? Imagina que tenemos los siguientes datos:

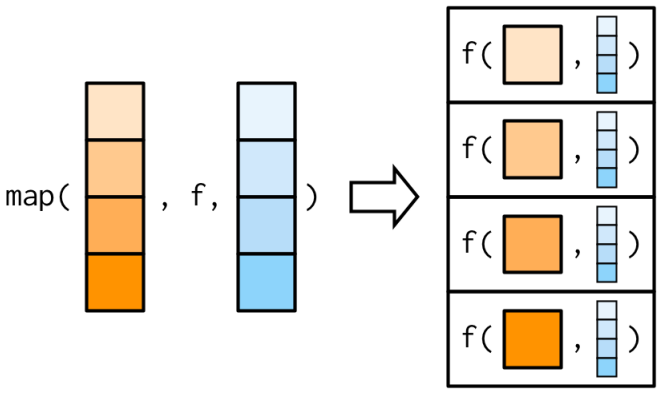
```
xs <- map(1:8, ~ runif(10))
xs[[1]][[1]] <- NA
ws <- map(1:8, ~ rpois(10, 5) + 1)
```

Puedes usar `map_dbl()` para calcular las medias no ponderadas:

```
map_dbl(xs, mean)
#> [1]    NA 0.463 0.551 0.453 0.564 0.501 0.371 0.443
```

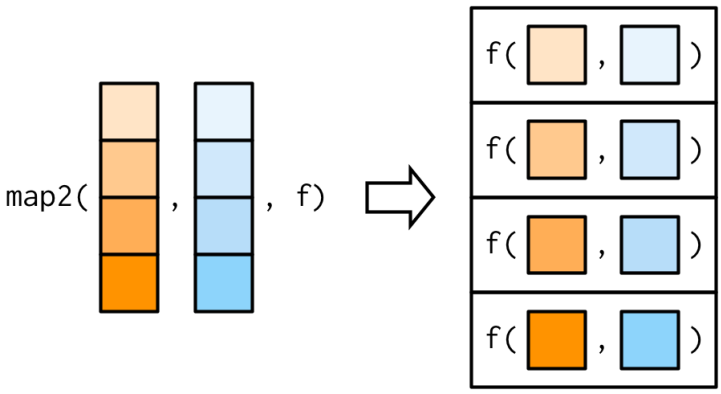
Pero pasar `ws` como argumento adicional no funciona porque los argumentos después de `.f` no se transforman:

```
map_dbl(xs, weighted.mean, w = ws)
#> Error in `map_dbl()`:
#>   In index: 1.
#> Caused by error in `weighted.mean.default()`:
#> ! 'x' and 'w' must have the same length
```



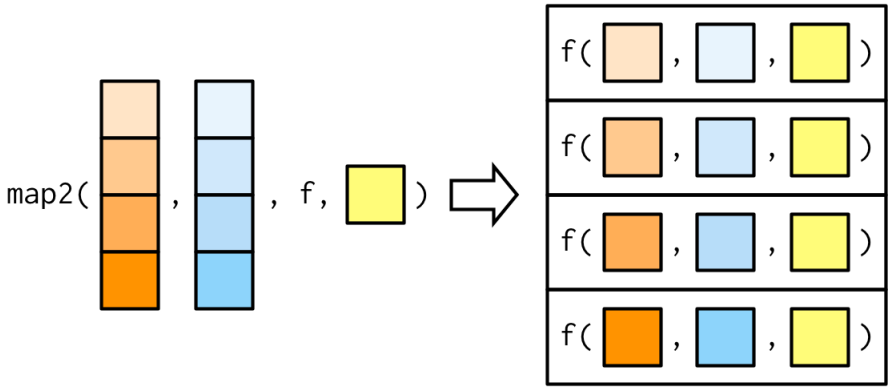
Necesitamos una nueva herramienta: un `map2()`, que se vectoriza sobre dos argumentos. Esto significa que tanto `.x` como `.y` varían en cada llamada a `.f`:

```
map2_dbl(xs, ws, weighted.mean)
#> [1]    NA 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```



Los argumentos de `map2()` son ligeramente diferentes a los argumentos de `map()` ya que dos vectores vienen antes de la función, en lugar de uno. Los argumentos adicionales todavía van después:

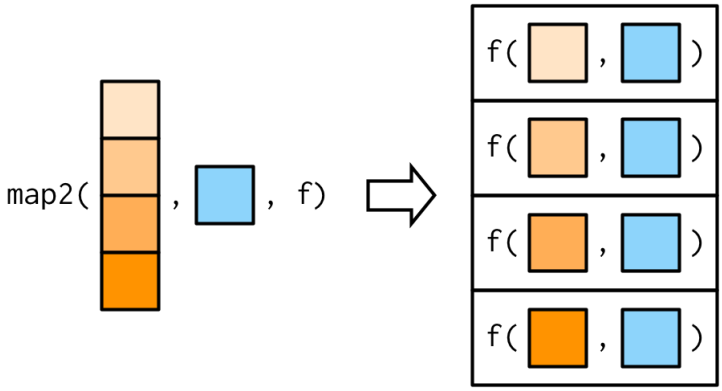
```
map2_dbl(xs, ws, weighted.mean, na.rm = TRUE)
#> [1] 0.504 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```



La implementación básica de `map2()` es simple y bastante similar a la de `map()`. En lugar de iterar sobre un vector, iteramos sobre dos en paralelo:

```
simple_map2 <- function(x, y, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], y[[i]], ...)
  }
  out
}
```

Una de las grandes diferencias entre `map2()` y la función simple anterior es que `map2()` recicla sus entradas para asegurarse de que tengan la misma longitud:



En otras palabras, `map2(x, y, f)` automáticamente se comportará como `map(x, f, y)` cuando sea necesario. Esto es útil al escribir funciones; en las secuencias de comandos, generalmente solo usaría la forma más simple directamente.

La base equivalente más cercana a `map2()` es `Map()`, que se analiza en la Section 9.4.5.

9.4.3. Sin salidas: `walk()` y amigos

La mayoría de las funciones se llaman por el valor que devuelven, por lo que tiene sentido capturar y almacenar el valor con una función `map()`. Pero algunas funciones se llaman principalmente por sus efectos secundarios (por ejemplo, `cat()`, `write.csv()` o `ggsave()`) y no tiene sentido capturar sus resultados. Toma este ejemplo simple que muestra un mensaje de bienvenida usando `cat()`. `cat()` devuelve `NULL`, así que mientras `map()` funciona (en el sentido de que genera las bienvenidas deseadas), también devuelve `list(NULL, NULL)`.

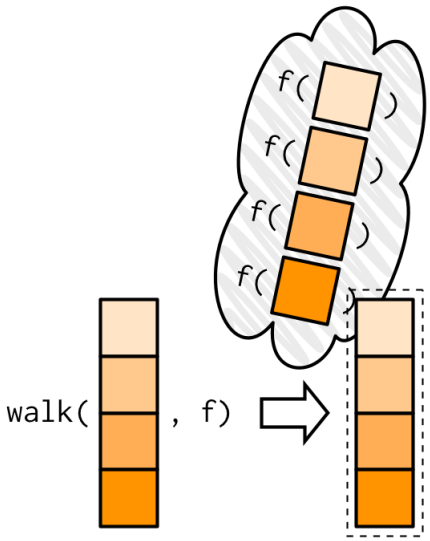
```
welcome <- function(x) {
  cat("Welcome ", x, "!\n", sep = "")
}
names <- c("Hadley", "Jenny")

# Además de generar las bienvenidas, también muestra
# el valor de retorno de cat()
map(names, welcome)
#> Welcome Hadley!
#> Welcome Jenny!
#> [[1]]
#> NULL
#>
#> [[2]]
#> NULL
```

Podrías evitar este problema asignando los resultados de `map()` a una variable que nunca usas, pero que enturbiaría la intención del código. En su lugar, purrr proporciona la familia de funciones walk que ignoran los valores de retorno de `.f` y en su lugar devuelven `.x` de forma invisible[1].

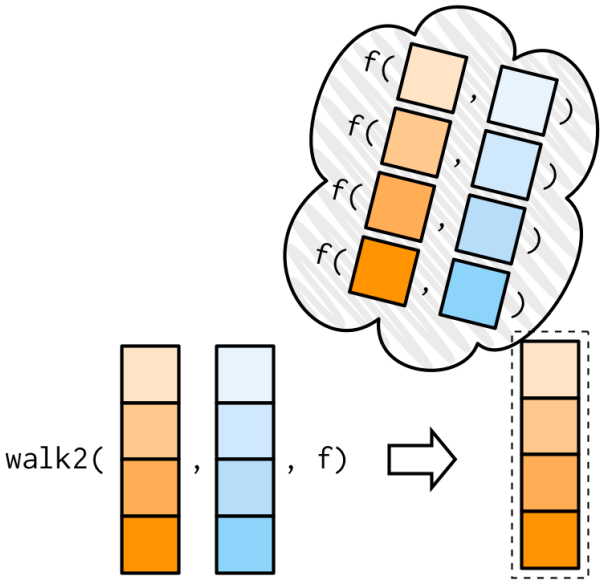
```
walk(names, welcome)
#> Welcome Hadley!
#> Welcome Jenny!
```

Mi representación visual de caminar intenta capturar la importante diferencia con `map()`: las salidas son efímeras y la entrada se devuelve de forma invisible.

[1] En resumen, los valores invisibles solo se imprimen si lo solicita explícitamente. Esto los hace muy adecuados para las funciones llamadas principalmente por sus efectos secundarios, ya que permite ignorar su salida de forma predeterminada, al tiempo que ofrece una opción para capturarla. Ver Section 6.7.2 para más detalles.



Una de las variantes de `walk()` más útiles es `walk2()` porque un efecto secundario muy común es guardar algo en el disco, y cuando guardas algo en el disco siempre tienes un par de valores: el objeto y la ruta que en el que desea guardarlo.



Por ejemplo, imagina que tienes una lista de data frames (que he creado aquí usando `split()`) y te gustaría guardar cada uno en un archivo CSV separado. Eso es fácil con `walk2()`:

```
temp <- tempfile()
dir.create(temp)

cyls <- split(mtcars, mtcars$cyl)
paths <- file.path(temp, paste0("cyl-", names(cyls), ".csv"))
walk2(cyls, paths, write.csv)

dir(temp)
#> [1] "cyl-4.csv" "cyl-6.csv" "cyl-8.csv"
```

Aquí el `walk2 ()` es equivalente a `write.csv(cyls[[1]], paths[[1]])`, `write.csv(cyls[[2]], paths[[2]])`, `write.csv(cyls[[3]], paths[[3]])`.

No existe una base equivalente a `walk()`; envuelva el resultado de `lapply()` en `invisible()` o guárdelo en una variable que nunca se use.

9.4.4. Iterando sobre valores e índices

Hay tres formas básicas de recorrer un vector con un bucle for:

- Bucle sobre los elementos: `for (x in xs)`
- Bucle sobre los índices numéricos: `for (i in seq_along(xs))`
- Bucle sobre los nombres: `for (nm in names(xs))`

La primera forma es análoga a la familia `map()`. Las formas segunda y tercera son equivalentes a la familia `imap()` que le permite iterar sobre los valores y los índices de un vector en paralelo.

`imap()` es como `map2()` en el sentido de que su `.f` se llama con dos argumentos, pero aquí ambos se derivan del vector. `imap(x, f)` es equivalente a `map2(x, nombres(x), f)` si x tiene nombres, y `map2(x, seq_along(x), f)` si no los tiene.

`imap()` suele ser útil para construir etiquetas:

```
imap_chr(iris, ~ paste0("The first value of ", .y, " is ", .x[[1]]))
#>                               Sepal.Length
#> "The first value of Sepal.Length is 5.1"
#>                                Sepal.Width
#>  "The first value of Sepal.Width is 3.5"
#>                               Petal.Length
#> "The first value of Petal.Length is 1.4"
#>                                Petal.Width
#>  "The first value of Petal.Width is 0.2"
#>                                    Species
#>   "The first value of Species is setosa"
```

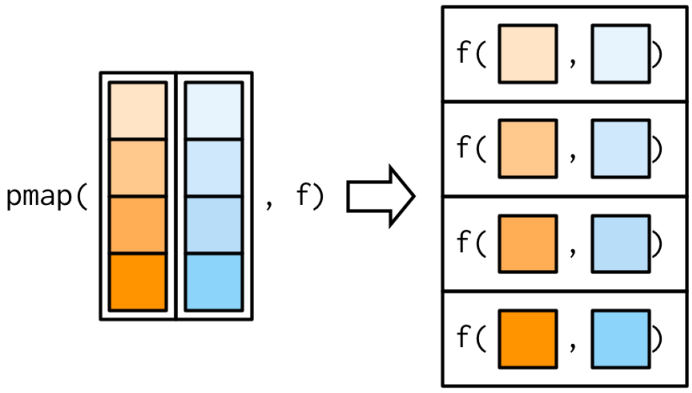
Si el vector no tiene nombre, el segundo argumento será el índice:

```
x <- map(1:6, ~ sample(1000, 10))
imap_chr(x, ~ paste0("The highest value of ", .y, " is ", max(.x)))
#> [1] "The highest value of 1 is 975" "The highest value of 2 is 915"
#> [3] "The highest value of 3 is 982" "The highest value of 4 is 955"
#> [5] "The highest value of 5 is 971" "The highest value of 6 is 696"
```

`imap()` es una ayuda útil si desea trabajar con los valores de un vector junto con sus posiciones.

9.4.5. Cualquier número de entradas: `pmap ()` y amigos

Ya que tenemos `map()` y `map2()`, podrías esperar `map3()`, `map4()`, `map5()`, … Pero, ¿dónde te detendrías? En lugar de generalizar `map2()` a un número arbitrario de argumentos, purrr adopta un rumbo ligeramente diferente con `pmap()`: le proporciona una sola lista, que contiene cualquier número de argumentos. En la mayoría de los casos, será una lista de vectores de igual longitud, es decir, algo muy similar a un data frame. En los diagramas, enfatizaré esa relación dibujando la entrada de forma similar a un data frame.

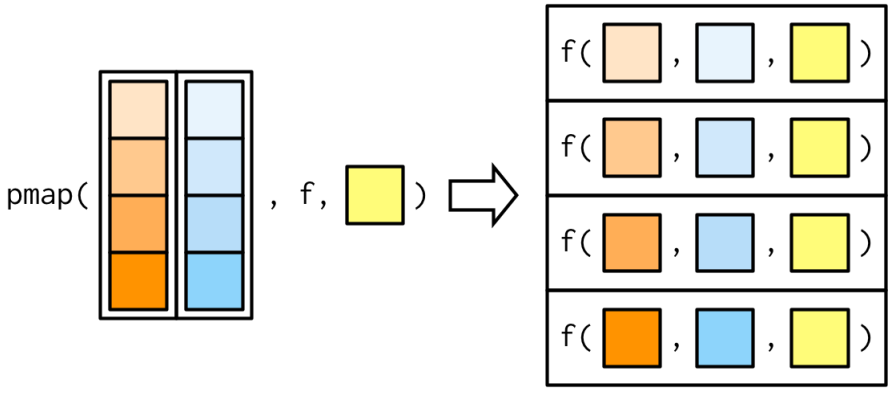


Hay una equivalencia simple entre `map2()` y `pmap()`: `map2(x, y, f)` es lo mismo que `pmap(list(x, y), f)`. El `pmap()` equivalente a `map2_dbl(xs, ws,weighted.mean)` utilizado anteriormente es:

```
pmap_dbl(list(xs, ws), weighted.mean)
#> [1]    NA 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```

Como antes, los argumentos variables vienen antes de `.f` (aunque ahora deben estar envueltos en una lista), y los argumentos constantes vienen después.

```
pmap_dbl(list(xs, ws), weighted.mean, na.rm = TRUE)
#> [1] 0.504 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```



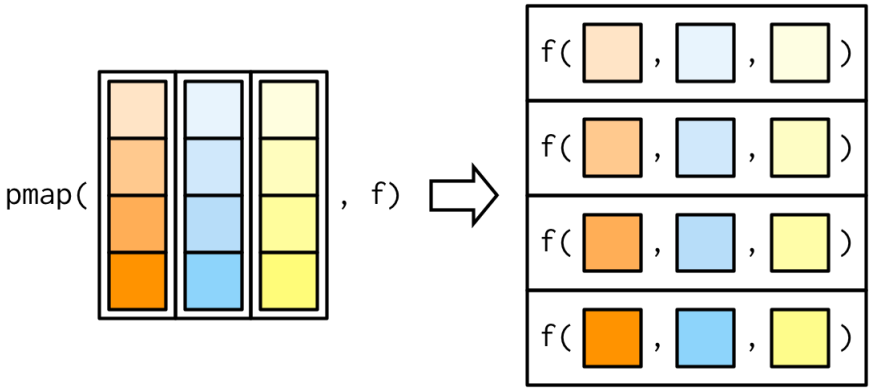
Una gran diferencia entre `pmap()` y las otras funciones de mapa es que `pmap()` te da un control mucho más preciso sobre la coincidencia de argumentos porque puedes nombrar los componentes de la lista. Volviendo a nuestro ejemplo de la Section 9.2.5, donde queríamos cambiar el argumento `trim` a `x`, podríamos usar `pmap()`:

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(1000)

pmap_dbl(list(trim = trims), mean, x = x)
#> [1] -6.6740  0.0210  0.0235  0.0151
```

Creo que es una buena práctica nombrar los componentes de la lista para dejar muy claro cómo se llamará a la función.

A menudo es conveniente llamar a `pmap()` con un data frame. Una forma práctica de crear ese data frame es con `tibble::tribble()`, que le permite describir un data frame fila por fila (en lugar de columna por columna, como de costumbre): pensando en los parámetros a una función como data frame es un patrón muy poderoso. El siguiente ejemplo muestra cómo puede dibujar números uniformes aleatorios con diferentes parámetros:

```
params <- tibble::tribble(
  ~ n, ~ min, ~ max,
   1L,     0,     1,
   2L,    10,   100,
   3L,   100,  1000
)

pmap(params, runif)
#> [[1]]
#> [1] 0.332
#>
#> [[2]]
#> [1] 53.5 47.6
#>
#> [[3]]
#> [1] 231 715 515
```



Aquí, los nombres de las columnas son fundamentales: elegí cuidadosamente hacerlos coincidir con los argumentos de `runif()`, por lo que `pmap(params, runif)` es equivalente a `runif(n = 1L, min = 0, max = 1)`, `runif(n = 2, min = 10, max = 100)`, `runif(n = 3L, min = 100, max = 1000)`. (Si tiene un data frame en la mano y los nombres no coinciden, use `dplyr::rename()` o similar).

Hay dos equivalentes base para la familia `pmap()`: `Map()` y `mapply()`. Ambos tienen importantes inconvenientes:

- `Map()` vectoriza sobre todos los argumentos para que no pueda proporcionar argumentos que no varíen.
- `mapply()` es la versión multidimensional de `sapply()`; conceptualmente, toma la salida de `Map()` y la simplifica si es posible. Esto le da problemas similares a `sapply()`. No existe un equivalente de múltiples entradas de `vapply()`.

9.4.6. Ejercicios

1. Explique los resultados de `modify(mtcars, 1)`.

2. Reescribe el siguiente código para usar `iwalk()` en lugar de `walk2()`. ¿Cuáles son las ventajas y desventajas?

```
cyls <- split(mtcars, mtcars$cyl)
paths <- file.path(temp, paste0("cyl-", names(cyls), ".csv"))
walk2(cyls, paths, write.csv)
```

3. Explique cómo el siguiente código transforma un data frame utilizando funciones almacenadas en una lista.

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) factor(x, labels = c("auto", "manual"))
)

nm <- names(trans)
mtcars[nm] <- map2(trans, mtcars[nm], function(f, var) f(var))
```

   Compare y contraste el enfoque `map2()` con este enfoque `map()`:

```
mtcars[nm] <- map(nm, ~ trans[[.x]](mtcars[[.x]]))
```

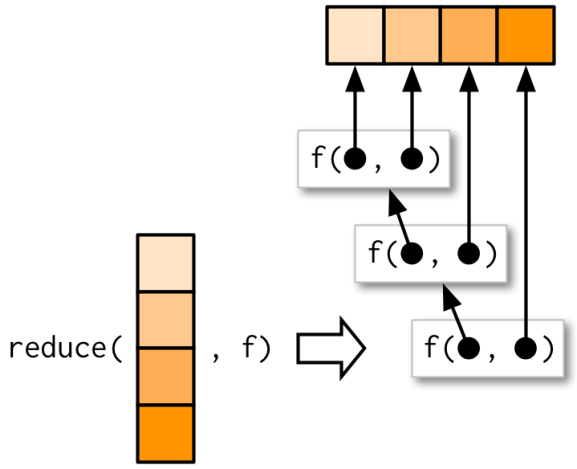
4. ¿Qué devuelve `write.csv()`, es decir, qué sucede si lo usa con `map2()` en lugar de `walk2()`?

9.5. Familia reduce

Después de la familia map, la siguiente familia de funciones más importante es la familia reduce. Esta familia es mucho más pequeña, con solo dos variantes principales, y se usa con menos frecuencia, pero es una idea poderosa, nos brinda la oportunidad de analizar algo de álgebra útil y potencia el marco de reducción de mapas que se usa con frecuencia para procesar conjuntos de datos muy grandes.

9.5.1. Lo esencial

`reduce()` toma un vector de longitud n y produce un vector de longitud 1 llamando a una función con un par de valores a la vez: `reduce(1:4, f)` es equivalente a `f(f(f(1, 2), 3), 4)`.



`reduce()` es una forma útil de generalizar una función que funciona con dos entradas (una función **binaria**) para que funcione con cualquier cantidad de entradas. Imagina que tienes una lista de vectores numéricos y quieres encontrar los valores que ocurren en cada elemento. Primero generamos algunos datos de muestra:

```
l <- map(1:4, ~ sample(1:10, 15, replace = T))
str(l)
#> List of 4
#>  $ : int [1:15] 7 1 8 8 3 8 2 4 7 10 ...
#>  $ : int [1:15] 3 1 10 2 5 2 9 8 5 4 ...
#>  $ : int [1:15] 6 10 9 5 6 7 8 6 10 8 ...
#>  $ : int [1:15] 9 8 6 4 4 5 2 9 9 6 ...
```

Para resolver este desafío necesitamos usar `intersect()` repetidamente:

```
out <- l[[1]]
out <- intersect(out, l[[2]])
out <- intersect(out, l[[3]])
out <- intersect(out, l[[4]])
out
#> [1] 8 4
```

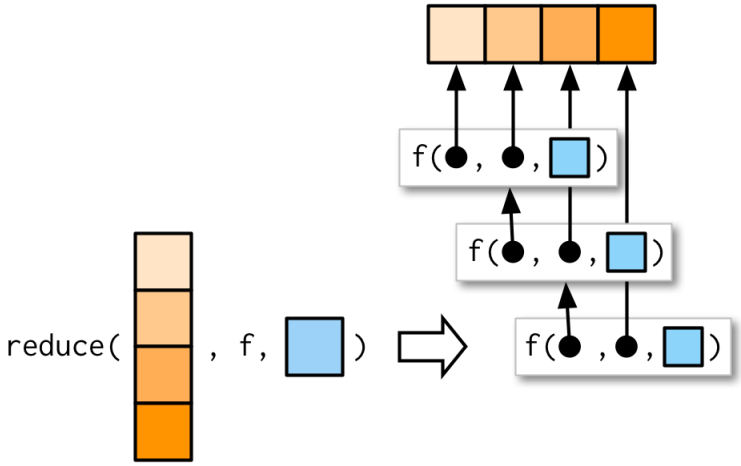
`reduce()` Automatiza esta solución para nosotros, para que podamos escribir:

```
reduce(l, intersect)
#> [1] 8 4
```

Podríamos aplicar la misma idea si quisiéramos listar todos los elementos que aparecen en al menos una entrada. Todo lo que tenemos que hacer es cambiar de `intersect()` a `union()`:

```
reduce(l, union)
#>  [1]  7  1  8  3  2  4 10  5  9  6
```

Al igual que la familia de mapas, también puede pasar argumentos adicionales. `intersect()` y `union()` no aceptan argumentos adicionales, así que no puedo demostrarlos aquí, pero el principio es sencillo y le hice un dibujo.



Como de costumbre, la esencia de `reduce()` se puede reducir a un simple envoltorio alrededor de un bucle for:

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

El equivalente básico es `Reduce()`. Tenga en cuenta que el orden de los argumentos es diferente: la función viene primero, seguida del vector, y no hay forma de proporcionar argumentos adicionales.

9.5.2. accumulate

La primera variante `reduce()`, `accumulate()`, es útil para comprender cómo funciona reduce, porque en lugar de devolver solo el resultado final, también devuelve todos los resultados intermedios:

```
accumulate(l, intersect)
#> [[1]]
#>  [1]  7  1  8  8  3  8  2  4  7 10 10  3  7 10 10
#>
#> [[2]]
#> [1]  1  8  3  2  4 10
#>
#> [[3]]
#> [1]  8  4 10
#>
#> [[4]]
#> [1] 8 4
```

Otra forma útil de entender reduce es pensar en `sum()`: `sum(x)` es equivalente a `x[[1]] + x[[2]] + x[[3]] + ...`, es decir `reduce(x, `+`)`. Entonces `accumulate(x, `+`)` es la suma acumulada:

```
x <- c(4, 3, 10)
reduce(x, `+`)
#> [1] 17

accumulate(x, `+`)
#> [1]  4  7 17
```

9.5.3. Tipos de salida

En el ejemplo anterior usando `+`, ¿qué debería devolver `reduce()` cuando `x` es corto, es decir, longitud 1 o 0? Sin argumentos adicionales, `reduce()` solo devuelve la entrada cuando `x` tiene una longitud de 1:

```
reduce(1, `+`)
#> [1] 1
```

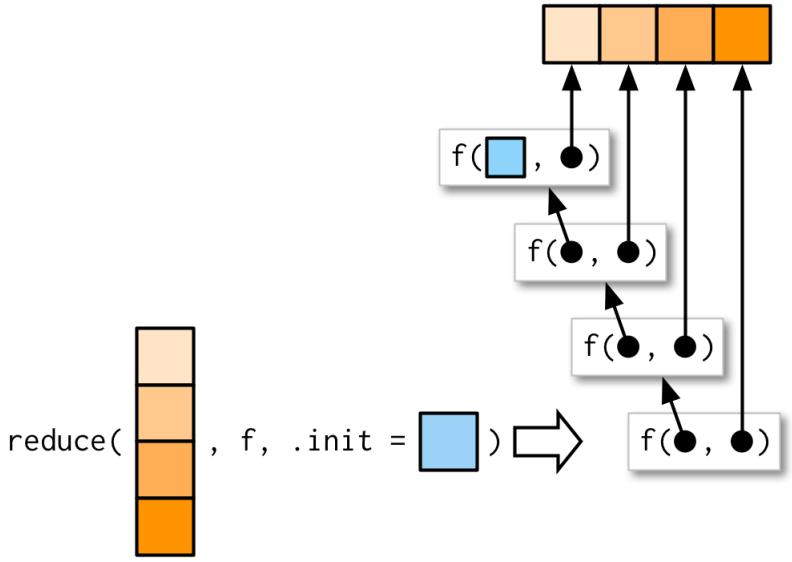
Esto significa que `reduce()` no tiene forma de verificar que la entrada sea válida:

```
reduce("a", `+`)
#> [1] "a"
```

¿Qué pasa si es de longitud 0? Recibimos un error que sugiere que necesitamos usar el argumento `.init`:

```
reduce(integer(), `+`)
#> Error in `reduce()`:
#> ! Must supply `.init` when `.x` is empty.
```

¿Qué debería ser `.init` aquí? Para averiguarlo, necesitamos ver qué sucede cuando se proporciona `.init`:



Así que si llamamos a `reduce(1, `+`, init)` el resultado será `1 + init`. Ahora sabemos que el resultado debería ser solo `1`, lo que sugiere que `.init` debería ser 0:

```
reduce(integer(), `+`, .init = 0)
#> [1] 0
```

Esto también asegura que `reduce()` verifique que las entradas de longitud 1 sean válidas para la función que estás llamando:

```
reduce("a", `+`, .init = 0)
#> Error in .x + .y: non-numeric argument to binary operator
```

Si quieres ser algebraico al respecto, 0 se llama la **identidad** de los números reales en la operación de suma: si agregas un 0 a cualquier número,

obtienes el mismo número. R aplica el mismo principio para determinar qué debe devolver una función de resumen con una entrada de longitud cero:

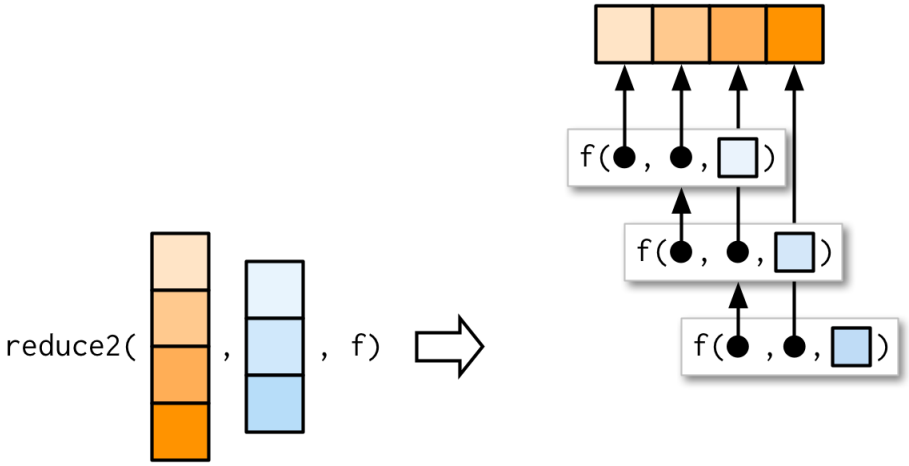
```
sum(integer())  # x + 0 = x
#> [1] 0
prod(integer()) # x * 1 = x
#> [1] 1
min(integer())  # min(x, Inf) = x
#> [1] Inf
max(integer())  # max(x, -Inf) = x
#> [1] -Inf
```

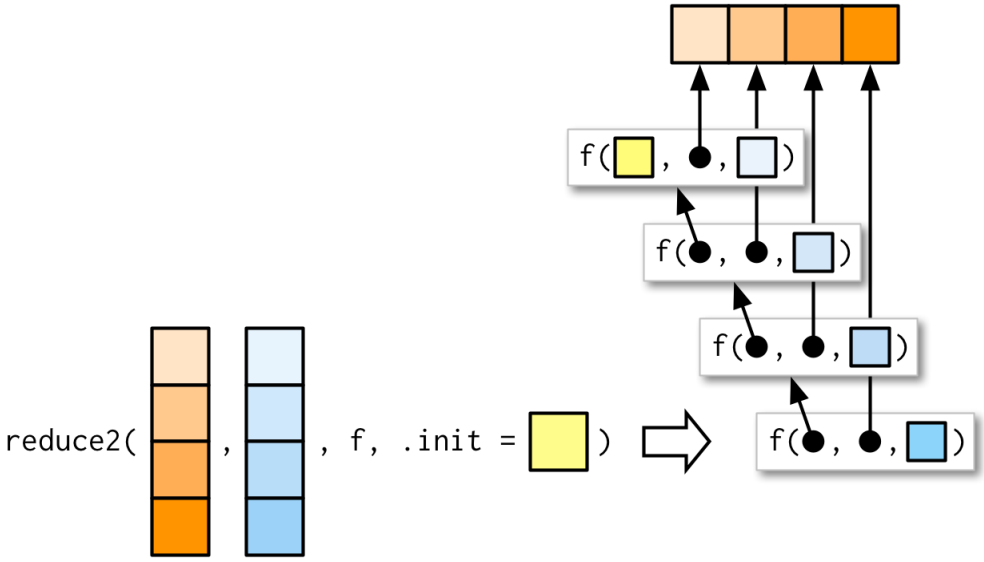
Si está utilizando `reduce()` en una función, siempre debe proporcionar `.init`. Piense detenidamente qué debe devolver su función cuando pasa un vector de longitud 0 o 1, y asegúrese de probar su implementación.

9.5.4. Múltiples entradas

Muy ocasionalmente necesita pasar dos argumentos a la función que está reduciendo. Por ejemplo, puede tener una lista de data frames que desea unir y las variables que usa para unir variarán de un elemento a otro. Este es un escenario muy especializado, por lo que no quiero dedicarle mucho tiempo, pero sí quiero que sepas que `reduce2()` existe.

La longitud del segundo argumento varía en función de si se proporciona `.init` o no: si tiene cuatro elementos de `x`, `f` solo se llamará tres veces. Si proporciona init, `f` se llamará cuatro veces.


reduce2(, , f)
f(, ,)
f(, ,)
f(, ,)


reduce2(, , f, .init =)
f(, ,)
f(, ,)
f(, ,)
f(, ,)

9.5.5. Mapa reducido

Es posible que haya oído hablar de map-reduce, la idea que impulsa la tecnología como Hadoop. Ahora puedes ver cuán simple y poderosa es la idea subyacente: map-reduce es un mapa combinado con una reducción. La diferencia para los datos grandes es que los datos se distribuyen en varias computadoras. Cada computadora realiza el mapa en los datos que tiene, luego envía el resultado a un coordinador que *reduce* los resultados individuales a un solo resultado.

Como un ejemplo simple, imagine calcular la media de un vector muy grande, tan grande que tiene que dividirse entre varias computadoras. Puede pedirle a cada computadora que calcule la suma y la longitud, y luego devolverlos al coordinador que calcula la media general dividiendo la suma total por la longitud total.

9.6. Funcionales de predicado

Un **predicado** es una función que devuelve un solo `TRUE` o `FALSE`, como `is.character()`, `is.null()` o `all()`, y decimos un predicado **coincide** con un vector si devuelve `TRUE`.

9.6.1. Lo esencial

Un **predicado funcional** aplica un predicado a cada elemento de un vector. purrr proporciona siete funciones útiles que se dividen en tres grupos:

- `some(.x, .p)` devuelve `TRUE` si *algún* elemento coincide;
  `every(.x, .p)` devuelve `TRUE` si *todos* los elementos coinciden;
  `none(.x, .p)` devuelve `TRUE` si *ningún* elemento coincide.

Estos son similares a `any(map_lgl(.x, .p))`, `all(map_lgl(.x, .p))` y `all(map_lgl(.x, negate(.p)))` pero terminar antes de tiempo: `some()` devuelve `TRUE` cuando ve el primer `TRUE`, y `cada()` y `ninguno()` devuelve `FALSE` cuando ven el primer `FALSE` o `TRUE` respectivamente .

- `detect(.x, .p)` devuelve el *valor* de la primera coincidencia; `detect_index(.x, .p)` devuelve la *ubicación* de la primera coincidencia.
- `keep(.x, .p)` *mantiene* todos los elementos coincidentes; `discard(.x, .p)` *suelta* todos los elementos coincidentes.

El siguiente ejemplo muestra cómo puede usar estas funciones con un data frame:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
detect(df, is.factor)
#> NULL
detect_index(df, is.factor)
#> [1] 0

str(keep(df, is.factor))
#> 'data.frame':    3 obs. of  0 variables
str(discard(df, is.factor))
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

9.6.2. Variantes de map

`map()` y `modify()` vienen en variantes que también toman funciones de predicado, transformando solo los elementos de `.x` donde `.p` es `TRUE`.

```
df <- data.frame(
  num1 = c(0, 10, 20),
  num2 = c(5, 6, 7),
  chr1 = c("a", "b", "c"),
  stringsAsFactors = FALSE
)

str(map_if(df, is.numeric, mean))
#> List of 3
#>  $ num1: num 10
#>  $ num2: num 6
#>  $ chr1: chr [1:3] "a" "b" "c"
str(modify_if(df, is.numeric, mean))
#> 'data.frame':    3 obs. of  3 variables:
#>  $ num1: num  10 10 10
#>  $ num2: num  6 6 6
#>  $ chr1: chr  "a" "b" "c"
str(map(keep(df, is.numeric), mean))
#> List of 2
#>  $ num1: num 10
#>  $ num2: num 6
```

9.6.3. Ejercicios

1. ¿Por qué `is.na()` no es una función de predicado? ¿Qué función base de R está más cerca de ser una versión predicada de `is.na()`?

2. `simple_reduce()` tiene un problema cuando `x` tiene una longitud de 0 o de 1. Describa el origen del problema y cómo podría solucionarlo.

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

```
    out <- f(out, x[[i]])
  }
  out
}
```

3. Implemente la función `span()` de Haskell: dada una lista `x` y una función de predicado `f`, `span(x, f)` devuelve la ubicación de la ejecución secuencial más larga de elementos donde el predicado es verdadero. (Sugerencia: puede encontrar útil `rle()`).

4. Implementar `arg_max()`. Debe tomar una función y un vector de entradas, y devolver los elementos de la entrada donde la función devuelve el valor más alto. Por ejemplo, `arg_max(-10:5, function(x) x ^ 2)` debería devolver -10. `arg_max(-5:5, function(x) x ^ 2)` debería devolver `c(-5, 5)`. Implemente también la función coincidente `arg_min()`.

5. La siguiente función escala un vector para que caiga en el rango [0, 1]. ¿Cómo lo aplicaría a cada columna de un data frame? ¿Cómo lo aplicaría a cada columna numérica en un data frame?

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

9.7. Funcionales base

Para terminar el capítulo, aquí ofrezco un resumen de importantes funciones base que no son miembros de las familias map, reduce o predicate y, por lo tanto, no tienen equivalente en purrr. Esto no quiere decir que no sean importantes, pero tienen un sabor más matemático o estadístico y, en general, son menos útiles en el análisis de datos.

9.7.1. Matrices y arreglos

`map()` y amigos están especializados para trabajar con vectores unidimensionales. `base::apply()` está especializado para trabajar con vectores bidimensionales y superiores, es decir, matrices y arreglos. Puede pensar en `apply()` como una operación que resume una matriz o conjunto al colapsar cada fila o columna en un solo valor. Tiene cuatro argumentos:

- `X`, la matriz o arreglo para resumir.
- `MARGIN`, un vector entero que da las dimensiones para resumir, 1 = filas, 2 = columnas, etc. (El nombre del argumento proviene de pensar en los márgenes de una distribución conjunta).
- `FUN`, una función de resumen.
- `...` otros argumentos pasan a `FUN`.

Un ejemplo típico de `apply()` se ve así

```
a2d <- matrix(1:20, nrow = 5)
apply(a2d, 1, mean)
#> [1]  8.5  9.5 10.5 11.5 12.5
apply(a2d, 2, mean)
#> [1]  3  8 13 18
```

Puede especificar múltiples dimensiones para `MARGIN`, lo cual es útil para arreglos de alta dimensión:

```
a3d <- array(1:24, c(2, 3, 4))
apply(a3d, 1, mean)
#> [1] 12 13
apply(a3d, c(1, 2), mean)
#>      [,1] [,2] [,3]
#> [1,]   10   12   14
#> [2,]   11   13   15
```

Hay dos advertencias para usar `apply()`:

- Al igual que `base::sapply()`, no tienes control sobre el tipo de salida; se simplificará automáticamente a una lista, matriz o vector. Sin embargo, generalmente usa `apply()` con matrices numéricas y una función de resumen numérico, por lo que es menos probable que encuentre un problema que con `sapply()`.

- `apply()` tampoco es idempotente en el sentido de que si la función de resumen es el operador de identidad, la salida no siempre es la misma que la entrada.

```
a1 <- apply(a2d, 1, identity)
identical(a2d, a1)
#> [1] FALSE

a2 <- apply(a2d, 2, identity)
identical(a2d, a2)
#> [1] TRUE
```

- Nunca uses `apply()` con un data frame. Siempre lo obliga a una matriz, lo que conducirá a resultados no deseados si su data frame contiene algo más que números.

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
apply(df, 2, mean)
#> Warning in mean.default(newX[, i], ...): argument is not numeric or
#> logical: returning NA
#> Warning in mean.default(newX[, i], ...): argument is not numeric or
#> logical: returning NA
#>  x  y
#> NA NA
```

9.7.2. Preocupaciones matemáticas

Los funcionales son muy comunes en matemáticas. El límite, el máximo, las raíces (el conjunto de puntos donde `f(x) = 0`) y la integral definida son todos funcionales: dada una función, devuelven un solo número (o vector de números). A primera vista, estas funciones no parecen encajar con el tema de la eliminación de bucles, pero si profundiza, descubrirá que todas se implementan mediante un algoritmo que implica iteración.

Base R proporciona un conjunto útil:

- `integrate()` encuentra el área bajo la curva definida por `f()`
- `uniroot()` encuentra donde `f()` llega a cero
- `optimise()` encuentra la ubicación del valor más bajo (o más alto) de `f()`

El siguiente ejemplo muestra cómo se pueden usar los funcionales con una función simple, `sin()`:

```
integrate(sin, 0, pi)
#> 2 with absolute error < 2.2e-14
str(uniroot(sin, pi * c(1 / 2, 3 / 2)))
#> List of 5
#>  $ root      : num 3.14
#>  $ f.root    : num 1.22e-16
#>  $ iter      : int 2
#>  $ init.it   : int NA
#>  $ estim.prec: num 6.1e-05
str(optimise(sin, c(0, 2 * pi)))
#> List of 2
#>  $ minimum  : num 4.71
#>  $ objective: num -1
str(optimise(sin, c(0, pi), maximum = TRUE))
#> List of 2
```

```
#>  $ maximum  : num 1.57
#>  $ objective: num 1
```

9.7.3. Ejercicios

1. ¿Cómo organiza `apply()` la salida? Lea la documentación y realice algunos experimentos.
2. ¿Qué hacen `eapply()` y `rapply()`? ¿En purrr tiene equivalentes?
3. Desafío: lea sobre el algoritmo de punto fijo. Completa los ejercicios usando R.

10. Fábricas de funciones

10.1. Introducción

Una **fábrica de funciones** es una función que hace funciones. Aquí hay un ejemplo muy simple: usamos una fábrica de funciones (`power1()`) para hacer dos funciones secundarias (`square()` y `cube()`):

```
power1 <- function(exp) {
  function(x) {
    x ^ exp
  }
}

square <- power1(2)
cube <- power1(3)
```

No se preocupe si esto aún no tiene sentido, ¡debería tenerlo al final del capítulo!

Llamaré `square()` y `cube()` **funciones fabricadas**, pero este es solo un término para facilitar la comunicación con otros humanos: desde la perspectiva de R, no son diferentes a las funciones creadas de otra manera.

```
square(3)
#> [1] 9
cube(3)
#> [1] 27
```

Ya ha aprendido acerca de los componentes individuales que hacen posibles las fábricas de funciones:

- En la Section 6.2.3, aprendiste sobre las funciones de primera clase de R. En R, vinculas una función a un nombre de la misma manera que vinculas cualquier objeto a un nombre: con `<-`.
- En la Section 7.4.2, aprendiste que una función captura (encierra) el entorno en el que se crea.
- En la Section 7.4.4, aprendió que una función crea un nuevo entorno de ejecución cada vez que se ejecuta. Este entorno suele ser efímero, pero aquí se convierte en el entorno envolvente de la función fabricada.

En este capítulo, aprenderá cómo la combinación no obvia de estas tres funciones conduce a la fábrica de funciones. También verá ejemplos de su uso en visualización y estadísticas.

De las tres principales herramientas de programación funcional (funcionales, fábricas de funciones y operadores de funciones), las fábricas de funciones son las menos utilizadas. En general, no tienden a reducir la complejidad general del código, sino que dividen la complejidad en fragmentos más fáciles de digerir. Las fábricas de funciones también son un bloque de construcción importante para los muy útiles operadores de funciones, sobre los cuales aprenderá en el Chapter 11.

Estructura

- La Section 10.2 comienza el capítulo con una explicación de cómo funcionan las fábricas de funciones, reuniendo ideas del alcance y los entornos. También verá cómo se pueden usar fábricas de funciones para implementar una memoria para funciones, lo que permite que los datos persistan entre llamadas.

- La Section 10.3 ilustra el uso de fábricas de funciones con ejemplos de ggplot2. Verá dos ejemplos de cómo funciona ggplot2 con fábricas de funciones proporcionadas por el usuario y un ejemplo de cómo ggplot2 usa una fábrica de funciones internamente.
- La Section 10.4 utiliza fábricas de funciones para abordar tres desafíos de las estadísticas: comprender la transformación de Box-Cox, resolver problemas de máxima verosimilitud y dibujar remuestreos de arranque.
- La Section 10.5 muestra cómo puede combinar fábricas de funciones y funcionales para generar rápidamente una familia de funciones a partir de datos.

Requisitos previos

Asegúrese de estar familiarizado con el contenido de las Secciones Section 6.2.3 (funciones de primera clase), Section 7.4.2 (el entorno funcional) y Section 7.4.4 (entornos de ejecución) mencionados anteriormente.

Las fábricas de funciones solo necesitan base R. Usaremos un poco de rlang para mirar dentro de ellas más fácilmente, y usaremos ggplot2 y scales para explorar el uso de fábricas de funciones en la visualización.

```
library(rlang)
library(ggplot2)
library(scales)
```

10.2. Fundamentos de fábrica

La idea clave que hace que las fábricas de funciones funcionen se puede expresar de manera muy concisa:

> El entorno envolvente de la función fabricada es un entorno de ejecución de la fábrica de funciones.

Solo se necesitan unas pocas palabras para expresar estas grandes ideas, pero se necesita mucho más trabajo para entender realmente lo que esto significa. Esta sección te ayudará a juntar las piezas con exploración interactiva y algunos diagramas.

10.2.1. Entornos

Empecemos echando un vistazo a `square()` y `cube()`:

```
square
#> function(x) {
#>     x ^ exp
#>   }
#> <environment: 0x556ca7bfc460>

cube
#> function(x) {
#>     x ^ exp
#>   }
#> <bytecode: 0x556ca64465c8>
#> <environment: 0x556ca7b8c348>
```

Es obvio de dónde viene `x`, pero ¿cómo encuentra R el valor asociado con `exp`? La simple impresión de las funciones fabricadas no es reveladora porque los cuerpos son idénticos; los contenidos del entorno envolvente son los factores importantes. Podemos obtener un poco más de información usando `rlang::env_print()`. Eso nos muestra que tenemos dos entornos diferentes (cada uno de los cuales era originalmente un entorno de ejecución de `power1()`). Los entornos tienen el mismo padre, que es el entorno envolvente de `power1()`, el entorno global.

```
env_print(square)
#> <environment: 0x556ca7bfc460>
#> Parent: <environment: global>
#> Bindings:
#> • exp: <dbl>

env_print(cube)
#> <environment: 0x556ca7b8c348>
#> Parent: <environment: global>
#> Bindings:
#> • exp: <dbl>
```

`env_print()` nos muestra que ambos entornos tienen un enlace a `exp`, pero queremos ver su valor [1]. Podemos hacerlo obteniendo primero el entorno de la función y luego extrayendo los valores:

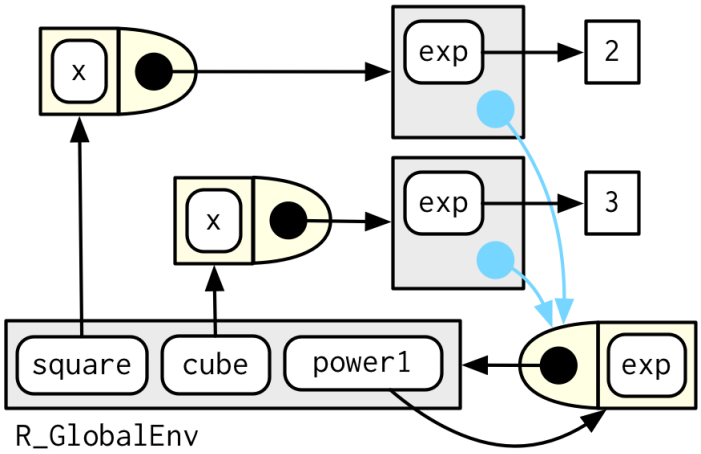
```
fn_env(square)$exp
#> [1] 2

fn_env(cube)$exp
#> [1] 3
```

Esto es lo que hace que las funciones fabricadas se comporten de manera diferente entre sí: los nombres en el entorno adjunto están vinculados a valores diferentes.

10.2.2. Convenciones de diagrama

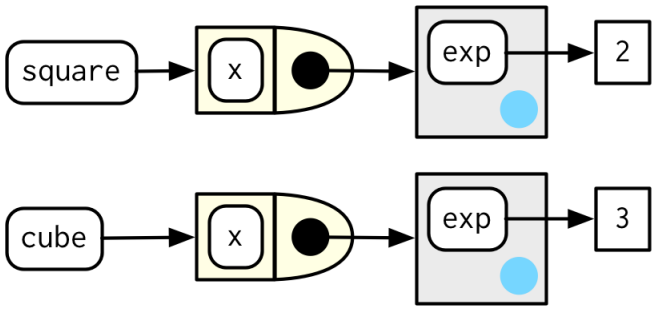
También podemos mostrar estas relaciones en un diagrama:

[1] Es probable que una versión futura de `env_print()` resuma mejor el contenido, por lo que no necesita este paso.



Están sucediendo muchas cosas en este diagrama y algunos de los detalles no son tan importantes. Podemos simplificar considerablemente usando dos convenciones:

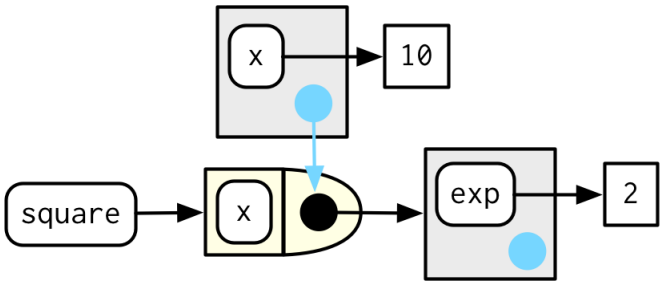
- Cualquier símbolo flotante libre vive en el entorno global.
- Cualquier entorno sin un padre explícito hereda del entorno global.



Esta vista, que se centra en los entornos, no muestra ningún vínculo directo entre `cube()` y `square()`. Esto se debe a que el enlace se realiza a través del cuerpo de la función, que es idéntico para ambos, pero no se muestra en este diagrama.

Para terminar, veamos el entorno de ejecución de `square(10)`. Cuando `square()` ejecuta `x ^ exp`, encuentra `x` en el entorno de ejecución y `exp` en su entorno adjunto.

```
square(10)
#> [1] 100
```



10.2.3. Evaluación forzada

Hay un error sutil en `power1()` causado por una evaluación perezosa. Para ver el problema necesitamos introducir alguna indirección:

```
x <- 2
square <- power1(x)
x <- 3
```

¿Qué debería devolver `square(2)`? Esperarías que devuelva 4:

```
square(2)
#> [1] 8
```

Desafortunadamente, no es así porque `x` solo se evalúa con pereza cuando se ejecuta `square()`, no cuando se ejecuta `power1()`. En general, este problema surgirá cada vez que cambie un enlace entre llamar a la función de fábrica y llamar a la función fabricada. Es probable que esto suceda rara vez, pero cuando sucede, conducirá a un verdadero error de rascado de cabeza.

Podemos solucionar este problema **forzando** la evaluación con `force()`:

```
power2 <- function(exp) {
  force(exp)
  function(x) {
    x ^ exp
  }
}

x <- 2
square <- power2(x)
x <- 3
square(2)
#> [1] 4
```

Cada vez que cree una fábrica de funciones, asegúrese de que se evalúen todos los argumentos, usando `force()` según sea necesario si el argumento solo lo usa la función fabricada.

10.2.4. Funciones con estado

Las fábricas de funciones también le permiten mantener el estado a través de las invocaciones de funciones, lo que generalmente es difícil de hacer

debido al principio de nuevo comienzo descrito en la Section 6.4.3.

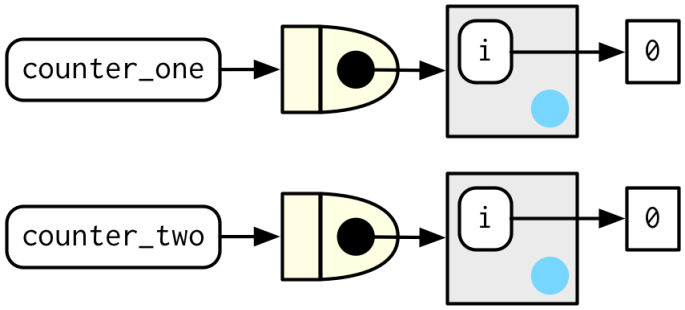
Hay dos cosas que lo hacen posible:

- El entorno envolvente de la función fabricada es único y constante.
- R tiene un operador de asignación especial, `<<-`, que modifica los enlaces en el entorno envolvente.

El operador de asignación habitual, `<-`, siempre crea un enlace en el entorno actual. El **operador de superasignación**, `<<-` vuelve a vincular un nombre existente que se encuentra en un entorno principal.

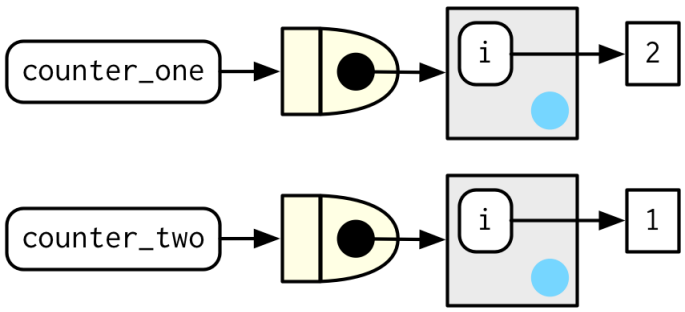
El siguiente ejemplo muestra cómo podemos combinar estas ideas para crear una función que registre cuántas veces ha sido llamada:

```
new_counter <- function() {
  i <- 0

  function() {
    i <<- i + 1
    i
  }
}

counter_one <- new_counter()
counter_two <- new_counter()
```



Cuando se ejecuta la función fabricada, `i <<- i + 1` modificará `i` en su entorno adjunto. Debido a que las funciones fabricadas tienen entornos envolventes independientes, tienen recuentos independientes:

```
counter_one()
#> [1] 1
counter_one()
#> [1] 2
counter_two()
#> [1] 1
```



Las funciones con estado se utilizan mejor con moderación. Tan pronto

como su función comience a administrar el estado de múltiples variables, es mejor cambiar a R6, el tema del Chapter 14.

10.2.5. Recolección de basura

Con la mayoría de las funciones, puede confiar en el recolector de basura para limpiar cualquier objeto temporal grande creado dentro de una función. Sin embargo, las funciones fabricadas se aferran al entorno de ejecución, por lo que deberá desvincular explícitamente cualquier objeto temporal grande con `rm()`. Compara los tamaños de `g1()` y `g2()` en el siguiente ejemplo:

```
f1 <- function(n) {
  x <- runif(n)
  m <- mean(x)
  function() m
}

g1 <- f1(1e6)
lobstr::obj_size(g1)
#> 8.01 MB

f2 <- function(n) {
  x <- runif(n)
  m <- mean(x)
  rm(x)
  function() m
}

g2 <- f2(1e6)
lobstr::obj_size(g2)
#> 12.96 kB
```

10.2.6. Ejercicios

1. La definición de `force()` es simple:

```
force
#> function (x)
#> x
#> <bytecode: 0x556ca4400d00>
#> <environment: namespace:base>
```

   Why is it better to `force(x)` instead of just `x`?

2. Base R contiene dos fábricas de funciones, `approxfun()` y `ecdf()`. Lea su documentación y experimente para descubrir qué hacen las funciones y qué devuelven.

3. Cree una función `pick()` que tome un índice, `i`, como argumento y devuelva una función con un argumento `x` que subjunte `x` con `i`.

```
pick(1)(x)
# should be equivalent to
x[[1]]

lapply(mtcars, pick(5))
# should be equivalent to
lapply(mtcars, function(x) x[[5]])
```

4. Cree una función que cree funciones que calculen el i^{th} momento central de un vector numérico. Puedes probarlo ejecutando el siguiente código:

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

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

5. ¿Qué pasa si no usas un cierre? Haz predicciones, luego verifica con el siguiente código.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
```

6. ¿Qué sucede si usa `<-` en lugar de `<<-`? Haz predicciones, luego verifica con el siguiente código.

```
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

10.3. Fábricas gráficas

Comenzaremos nuestra exploración de fábricas de funciones útiles con algunos ejemplos de ggplot2.

10.3.1. Etiquetado

Uno de los objetivos del paquete scales es facilitar la personalización de las etiquetas en ggplot2. Proporciona muchas funciones para controlar los

detalles finos de ejes y leyendas. Las funciones del formateador[2] son una clase útil de funciones que facilitan el control de la aparición de roturas de ejes. El diseño de estas funciones inicialmente puede parecer un poco extraño: todas devuelven una función, a la que debe llamar para formatear un número.

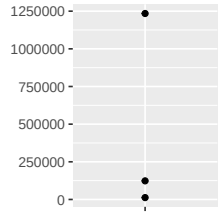
```
y <- c(12345, 123456, 1234567)
comma_format()(y)
#> [1] "12,345"    "123,456"   "1,234,567"

number_format(scale = 1e-3, suffix = " K")(y)
#> [1] "12 K"    "123 K"   "1 235 K"
```

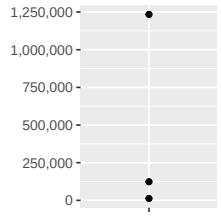
En otras palabras, la interfaz principal es una fábrica de funciones. A primera vista, esto parece agregar una complejidad adicional por poca ganancia. Pero permite una buena interacción con las escalas de ggplot2, porque aceptan funciones en el argumento `label`:

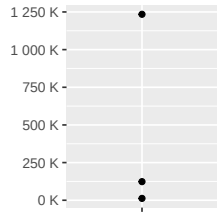
```
df <- data.frame(x = 1, y = y)
core <- ggplot(df, aes(x, y)) +
  geom_point() +
  scale_x_continuous(breaks = 1, labels = NULL) +
  labs(x = NULL, y = NULL)

core
core + scale_y_continuous(
  labels = comma_format()
)
core + scale_y_continuous(
  labels = number_format(scale = 1e-3, suffix = " K")
```

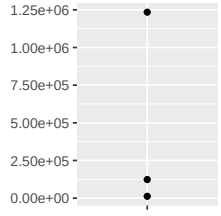
[2]Es un desafortunado accidente de la historia que las escalas usen sufijos de función en lugar de prefijos de función. Eso es porque fue escrito antes de que entendiera las ventajas de autocompletar al usar prefijos comunes en lugar de sufijos comunes.

```
)
core + scale_y_continuous(
  labels = scientific_format()
)
```







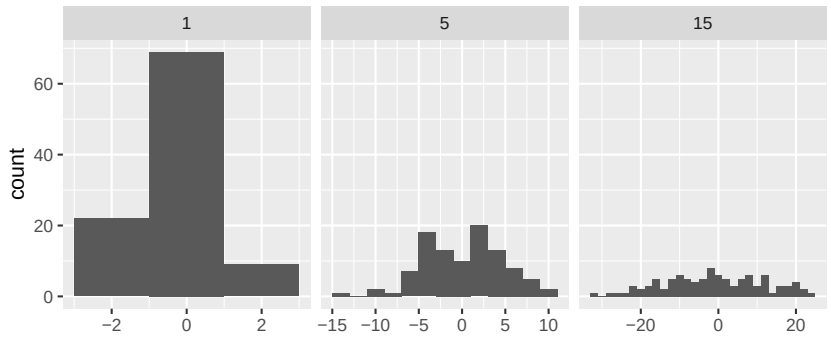


10.3.2. Contenedores de histograma

Una característica poco conocida de `geom_histogram()` es que el argumento `binwidth` puede ser una función. Esto es particularmente útil porque la función se ejecuta una vez para cada grupo, lo que significa que puede tener diferentes anchos de bin en diferentes facetas, lo que de otro modo no sería posible.

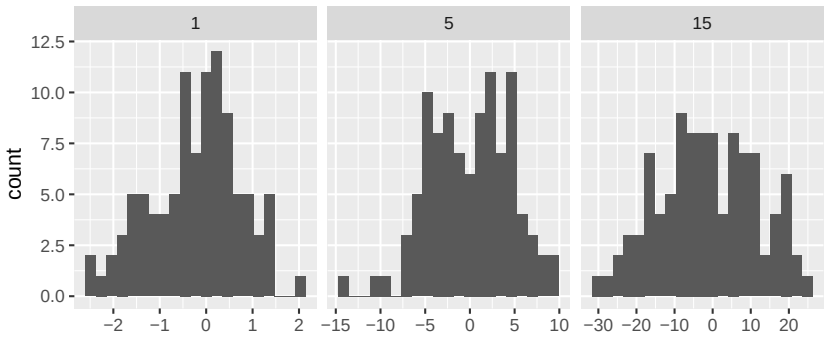
Para ilustrar esta idea y ver dónde podría ser útil el ancho de bin variable, voy a construir un ejemplo en el que un ancho de bin fijo no es muy bueno.

```
# construir algunos datos de muestra con números muy diferentes en cada celda
sd <- c(1, 5, 15)
n <- 100

df <- data.frame(x = rnorm(3 * n, sd = sd), sd = rep(sd, n))

ggplot(df, aes(x)) +
  geom_histogram(binwidth = 2) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```



Aquí cada faceta tiene el mismo número de observaciones, pero la variabilidad es muy diferente. Sería bueno si pudiéramos solicitar que los anchos de intervalo varíen para obtener aproximadamente el mismo número de observaciones en cada intervalo. Una forma de hacerlo es con una fábrica de funciones que ingresa el número deseado de contenedores (`n`) y genera una función que toma un vector numérico y devuelve un ancho de contenedor:

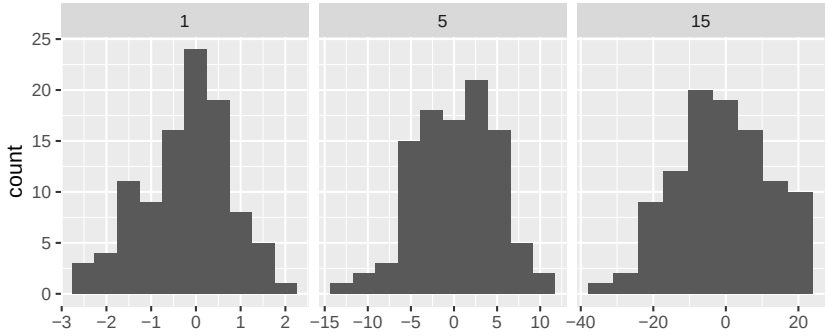
```
binwidth_bins <- function(n) {
  force(n)

  function(x) {
    (max(x) - min(x)) / n
  }
}

ggplot(df, aes(x)) +
  geom_histogram(binwidth = binwidth_bins(20)) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```



Podríamos usar este mismo patrón para envolver las funciones base

de R que automáticamente encuentran el llamado binwidth óptimo [3], `nclass.Sturges()`, `nclass.scott()` y `nclass .FD()`:

```
base_bins <- function(type) {
  fun <- switch(type,
    Sturges = nclass.Sturges,
    scott = nclass.scott,
    FD = nclass.FD,
    stop("Unknown type", call. = FALSE)
  )

  function(x) {
    (max(x) - min(x)) / fun(x)
  }
}

ggplot(df, aes(x)) +
  geom_histogram(binwidth = base_bins("FD")) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```



[3] ggplot2 no expone estas funciones directamente porque no creo que la definición de optimización necesaria para hacer que el problema sea matemáticamente tratable coincida con las necesidades reales de exploración de datos.

10.3.3. `ggsave()`

Finalmente, quiero mostrar una fábrica de funciones utilizada internamente por ggplot2. `ggplot2:::plot_dev()` es utilizado por `ggsave()` para pasar de una extensión de archivo (por ejemplo, `png`, `jpeg`, etc.) a una función de dispositivo gráfico (por ejemplo, `png()`, `jpeg( )`). El desafío aquí surge porque los dispositivos gráficos básicos tienen algunas inconsistencias menores que debemos corregir:

- La mayoría tiene `filename` como primer argumento, pero algunos tienen `file`.
- El `width` y `height` de los dispositivos gráficos de trama usan unidades de píxeles por defecto, pero los gráficos vectoriales usan pulgadas.

A continuación se muestra una versión levemente simplificada de `plot_dev()`:

```
plot_dev <- function(ext, dpi = 96) {
  force(dpi)

  switch(ext,
    eps =  ,
    ps  =  function(path, ...) {
      grDevices::postscript(
        file = filename, ..., onefile = FALSE,
        horizontal = FALSE, paper = "special"
      )
    },
    pdf = function(filename, ...) grDevices::pdf(file = filename, ...),
    svg = function(filename, ...) svglite::svglite(file = filename, ...),
    emf = ,
    wmf = function(...) grDevices::win.metafile(...),
```

```
    png = function(...) grDevices::png(..., res = dpi, units = "in"),
    jpg = ,
    jpeg = function(...) grDevices::jpeg(..., res = dpi, units = "in"),
    bmp = function(...) grDevices::bmp(..., res = dpi, units = "in"),
    tiff = function(...) grDevices::tiff(..., res = dpi, units = "in"),
    stop("Unknown graphics extension: ", ext, call. = FALSE)
  )
}

plot_dev("pdf")
#> function(filename, ...) grDevices::pdf(file = filename, ...)
#> <bytecode: 0x556cac322cf0>
#> <environment: 0x556cac0b0db0>
plot_dev("png")
#> function(...) grDevices::png(..., res = dpi, units = "in")
#> <bytecode: 0x556cac482b10>
#> <environment: 0x556cac6a56f0>
```

10.3.4. Ejercicios

1. Comparar y contrastar `ggplot2::label_bquote()` con `scales::number_format()`

10.4. Fábricas estadísticas

Los ejemplos más motivadores para las fábricas de funciones provienen de las estadísticas:

- La transformación Box-Cox.
- Remuestreo Bootstrap.
- Estimación de máxima verosimilitud.

Todos estos ejemplos se pueden abordar sin fábricas de funciones, pero creo que las fábricas de funciones son una buena opción para estos problemas y brindan soluciones elegantes. Estos ejemplos requieren algunos antecedentes estadísticos, así que siéntase libre de omitirlos si no tienen mucho sentido para usted.

10.4.1. La transformación Box-Cox

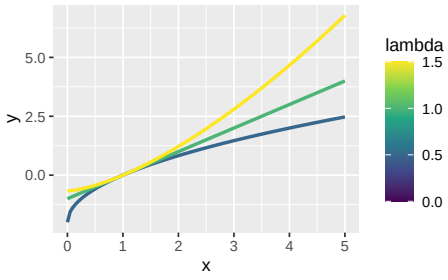
La transformación de Box-Cox (un tipo de transformación de potencia) es una transformación flexible que a menudo se usa para transformar los datos hacia la normalidad. Tiene un solo parámetro, λ, que controla la fuerza de la transformación. Podríamos expresar la transformación como una función simple de dos argumentos:

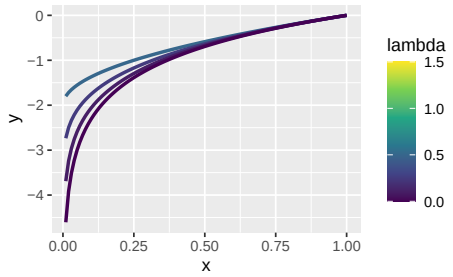
```
boxcox1 <- function(x, lambda) {
  stopifnot(length(lambda) == 1)

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

Pero volver a formular como una fábrica de funciones facilita la exploración de su comportamiento con `stat_function()`:

```
boxcox2 <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x ^ lambda - 1) / lambda
```

```
  }
}

stat_boxcox <- function(lambda) {
  stat_function(aes(colour = lambda), fun = boxcox2(lambda), size = 1)
}

ggplot(data.frame(x = c(0, 5)), aes(x)) +
  lapply(c(0.5, 1, 1.5), stat_boxcox) +
  scale_colour_viridis_c(limits = c(0, 1.5))
#> Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0
#>   Please use `linewidth` instead.

# visualmente, log() parece tener sentido como la transformación
# para lambda = 0; a medida que los valores se hacen cada vez más pequeños,
# se acerca cada vez más a una transformación de registro
ggplot(data.frame(x = c(0.01, 1)), aes(x)) +
  lapply(c(0.5, 0.25, 0.1, 0), stat_boxcox) +
  scale_colour_viridis_c(limits = c(0, 1.5))
```





En general, esto le permite usar una transformación de Box-Cox con cualquier función que acepte una función de transformación unaria: no tiene que preocuparse de que esa función proporcione `...` para pasar argumentos adicionales. También creo que la partición de `lambda` y `x` en dos argumentos de función diferentes es natural, ya que `lambda` juega un papel bastante diferente al de `x`.

10.4.2. Generadores de arranque

Las fábricas de funciones son un enfoque útil para el arranque. En lugar de pensar en un solo arranque (¡siempre necesita más de uno!), puede pensar en un **generador** de arranque, una función que produce un nuevo arranque cada vez que se llama:

```
boot_permute <- function(df, var) {
  n <- nrow(df)
  force(var)

  function() {
    col <- df[[var]]
    col[sample(n, replace = TRUE)]
  }
}
```

```
boot_mtcars1 <- boot_permute(mtcars, "mpg")
head(boot_mtcars1())
#> [1] 16.4 22.8 22.8 22.8 16.4 19.2
head(boot_mtcars1())
#> [1] 17.8 18.7 30.4 30.4 16.4 21.0
```

La ventaja de una fábrica de funciones es más clara con un bootstrap paramétrico donde primero tenemos que ajustar un modelo. Podemos hacer este paso de configuración una vez, cuando se llama a la fábrica, en lugar de una vez cada vez que generamos el arranque:

```
boot_model <- function(df, formula) {
  mod <- lm(formula, data = df)
  fitted <- unname(fitted(mod))
  resid <- unname(resid(mod))
  rm(mod)

  function() {
    fitted + sample(resid)
  }
}

boot_mtcars2 <- boot_model(mtcars, mpg ~ wt)
head(boot_mtcars2())
#> [1] 25.0 24.0 21.7 19.2 24.9 16.0
head(boot_mtcars2())
#> [1] 27.4 21.0 20.3 19.4 16.3 21.3
```

Uso `rm(mod)` porque los objetos del modelo lineal son bastante grandes (incluyen copias completas de la matriz del modelo y los datos de entrada) y quiero mantener la función fabricada lo más pequeña posible.

10.4.3. Estimación de máxima verosimilitud

El objetivo de la estimación de máxima verosimilitud (MLE) es encontrar los valores de los parámetros para una distribución que hacen que los datos observados sean más probables. Para hacer MLE, comienza con una función de probabilidad. Por ejemplo, tome la distribución de Poisson. Si conocemos λ, podemos calcular la probabilidad de obtener un vector $\mathbf{x}$ de valores (x_1, x_2, ..., x_n) multiplicando la función de probabilidad de Poisson como sigue:

$P(\lambda, \mathbf{x}) = \prod_{i=1}^{n} \frac{\lambda^{x_i} e^{-\lambda}}{x_i!}$

En estadística, casi siempre trabajamos con el registro de esta función. El logaritmo es una transformación monótona que conserva propiedades importantes (es decir, los extremos se encuentran en el mismo lugar), pero tiene ventajas específicas:

- El registro convierte un producto en una suma, con lo que es más fácil trabajar.
- Multiplicar números pequeños produce números aún más pequeños, lo que hace que la aproximación de punto flotante utilizada por una computadora sea menos precisa.

Apliquemos una transformación logarítmica a esta función de probabilidad y simplifiquemos tanto como sea posible:

$\log(P(\lambda, \mathbf{x})) = \sum_{i=1}^{n} \log(\frac{\lambda^{x_i} e^{-\lambda}}{x_i!})$

$\log(P(\lambda, \mathbf{x})) = \sum_{i=1}^{n} (x_i \log(\lambda) - \lambda - \log(x_i!))$

$\log(P(\lambda, \mathbf{x})) = \sum *i = 1^{n} x_i * \log(\lambda) - \sum i = 1^{n} \lambda - \sum_{i=1}^{n} \log(x_i!)$

$\log(P(\lambda, \mathbf{x})) = \log(\lambda) \sum *i = 1^{n} x_i - n * \lambda - \sum i = 1^{n} \log(x_i!)$

Ahora podemos convertir esta función en una función R. La función R es bastante elegante porque R está vectorizado y, dado que es un lenguaje

de programación estadístico, R viene con funciones integradas como log-factorial (`lfactorial()`).

```
lprob_poisson <- function(lambda, x) {
  n <- length(x)
  (log(lambda) * sum(x)) - (n * lambda) - sum(lfactorial(x))
}
```

Considere este vector de observaciones:

```
x1 <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
```

Podemos usar `lprob_poisson()` para calcular la probabilidad (registrada) de `x1` para diferentes valores de `lambda`.

```
lprob_poisson(10, x1)
#> [1] -184
lprob_poisson(20, x1)
#> [1] -61.1
lprob_poisson(30, x1)
#> [1] -31
```

Hasta ahora hemos estado pensando en `lambda` como fijo y conocido y la función nos dijo la probabilidad de obtener diferentes valores de `x`. Pero en la vida real, observamos 'x' y es 'lambda' lo que se desconoce. La verosimilitud es la función de probabilidad vista a través de este lente: queremos encontrar la `lambda` que hace que la `x` observada sea la más probable. Es decir, dada `x`, ¿qué valor de `lambda` nos da el valor más alto de `lprob_poisson`()?

En estadística, destacamos este cambio de perspectiva al escribir $f_{\mathbf{x}}(\lambda)$ en lugar de $f(\lambda, \mathbf{x})$. En R, podemos usar una fábrica de funciones. Proporcionamos `x` y generamos una función con un solo parámetro, `lambda`:

```
ll_poisson1 <- function(x) {
  n <- length(x)

  function(lambda) {
    log(lambda) * sum(x) - n * lambda - sum(lfactorial(x))
  }
}
```

(No necesitamos `force()` porque `length()` fuerza implícitamente la evaluación de `x`.)

Una cosa buena de este enfoque es que podemos hacer algunos cálculos previos: cualquier término que solo involucre `x` se puede calcular una vez en la fábrica. Esto es útil porque necesitaremos llamar a esta función muchas veces para encontrar la mejor `lambda`.

```
ll_poisson2 <- function(x) {
  n <- length(x)
  sum_x <- sum(x)
  c <- sum(lfactorial(x))

  function(lambda) {
    log(lambda) * sum_x - n * lambda - c
  }
}
```

Ahora podemos usar esta función para encontrar el valor de `lambda` que maximiza la probabilidad (log):

```
ll1 <- ll_poisson2(x1)

ll1(10)
#> [1] -184
```

```
ll1(20)
#> [1] -61.1
ll1(30)
#> [1] -31
```

En lugar de prueba y error, podemos automatizar el proceso de encontrar el mejor valor con `optimise()`. Evaluará `ll1()` muchas veces, utilizando trucos matemáticos para reducir el valor más grande lo más rápido posible. Los resultados nos dicen que el valor más alto es `-30.27` que ocurre cuando `lambda = 32.1`:

```
optimise(ll1, c(0, 100), maximum = TRUE)
#> $maximum
#> [1] 32.1
#>
#> $objective
#> [1] -30.3
```

Ahora, podríamos haber resuelto este problema sin usar una fábrica de funciones porque `optimise()` pasa `...` a la función que se está optimizando. Eso significa que podríamos usar la función de probabilidad de registro directamente:

```
optimise(lprob_poisson, c(0, 100), x = x1, maximum = TRUE)
#> $maximum
#> [1] 32.1
#>
#> $objective
#> [1] -30.3
```

La ventaja de usar una fábrica de funciones aquí es bastante pequeña, pero hay dos sutilezas:

- Podemos precalcular algunos valores en fábrica, ahorrando tiempo de cálculo en cada iteración.
- El diseño de dos niveles refleja mejor la estructura matemática del problema subyacente.

Estas ventajas aumentan en problemas MLE más complejos, donde tiene múltiples parámetros y múltiples vectores de datos.

10.4.4. Ejercicios

1. En `boot_model()`, ¿por qué no necesito forzar la evaluación de `df` o `formula`?
2. ¿Por qué podrías formular la transformación de Box-Cox de esta manera?

```
boxcox3 <- function(x) {
  function(lambda) {
    if (lambda == 0) {
      log(x)
    } else {
      (x ^ lambda - 1) / lambda
    }
  }
}
```

3. ¿Por qué no debe preocuparse de que `boot_permute()` almacene una copia de los datos dentro de la función que genera?
4. ¿Cuánto tiempo ahorra `ll_poisson2()` en comparación con `ll_poisson1()`? Use `bench::mark()` para ver cuánto más rápido ocurre la optimización. ¿Cómo cambia la longitud de 'x' los resultados?

10.5. Fábricas de funciones + funcionales

Para terminar el capítulo, mostraré cómo puede combinar funcionales y fábricas de funciones para convertir datos en muchas funciones. El siguiente código crea muchas funciones de potencia con nombres especiales al iterar sobre una lista de argumentos:

```
names <- list(
  square = 2,
  cube = 3,
  root = 1/2,
  cuberoot = 1/3,
  reciprocal = -1
)
funs <- purrr::map(names, power1)

funs$root(64)
#> [1] 8
funs$root
#> function(x) {
#>     x ^ exp
#>   }
#> <bytecode: 0x556ca64465c8>
#> <environment: 0x556cac936d08>
```

Esta idea se extiende de manera directa si su fábrica de funciones toma dos (reemplace `map()` con `map2()`) o más (reemplace con `pmap()`) argumentos.

Una desventaja de la construcción actual es que tienes que prefijar cada llamada de función con `funs$`. Hay tres formas de eliminar esta sintaxis adicional:

- Para un efecto muy temporal, puedes usar `with()`:

```
with(funs, root(100))
#> [1] 10
```

Recomiendo esto porque deja muy claro cuándo se ejecuta el código en un contexto especial y cuál es ese contexto.

- Para un efecto más prolongado, puede `attach()` las funciones a la ruta de búsqueda, luego `detach()` cuando haya terminado:

```
attach(funs)
#> The following objects are masked _by_ .GlobalEnv:
#>
#>     cube, square
root(100)
#> [1] 10
detach(funs)
```

Probablemente le hayan dicho que evite usar `attach()`, y ese es generalmente un buen consejo. Sin embargo, la situación es un poco diferente a lo habitual porque adjuntamos una lista de funciones, no un data frame. Es menos probable que modifique una función que una columna en un data frame, por lo que algunos de los peores problemas con `attach()` no se aplican.

- Finalmente, podrías copiar las funciones al entorno global con `env_bind()` (aprenderás sobre `!!!` en la Section 19.6). Esto es en su mayoría permanente:

```
rlang::env_bind(globalenv(), !!!funs)
root(100)
#> [1] 10
```

Más tarde puede desvincular esos mismos nombres, pero no hay garantía de que no se hayan vuelto a vincular mientras tanto, y es posible que esté eliminando un objeto que otra persona creó.

```
rlang::env_unbind(globalenv(), names(funs))
```

Aprenderá un enfoque alternativo para el mismo problema en la Section 19.7.4. En lugar de usar una fábrica de funciones, puede construir la función con cuasicomillas. Esto requiere conocimientos adicionales, pero genera funciones con cuerpos legibles y evita la captura accidental de objetos grandes en el alcance adjunto. Usamos esa idea en la Section 21.2.4 cuando trabajamos en herramientas para generar HTML desde R.

10.5.1. Ejercicios

1. ¿Cuál de los siguientes comandos es equivalente a `with(x, f(z))`?

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

2. Compare y contraste los efectos de `env_bind()` frente a `attach()` para el siguiente código.

```
funs <- list(
  mean = function(x) mean(x, na.rm = TRUE),
  sum = function(x) sum(x, na.rm = TRUE)
)

attach(funs)
#> The following objects are masked from package:base:
#>
#>     mean, sum
mean <- function(x) stop("Hi!")
detach(funs)
```

```
env_bind(globalenv(), !!!funs)
mean <- function(x) stop("Hi!")
env_unbind(globalenv(), names(funs))
```

11. Operadores de funciones

11.1. Introducción

En este capítulo, aprenderá acerca de los operadores de funciones. Un **operador de función** es una función que toma una (o más) funciones como entrada y devuelve una función como salida. El siguiente código muestra un operador de función simple, `chatty()`. Envuelve una función, creando una nueva función que imprime su primer argumento. Puede crear una función como esta porque le da una ventana para ver cómo funcionan las funciones, como `map_int()`.

```
chatty <- function(f) {
  force(f)

  function(x, ...) {
    message("Processing ", x)
    f(x, ...)
  }
}
f <- function(x) x ^ 2
s <- c(3, 2, 1)

purrr::map_dbl(s, chatty(f))
#> Processing 3
#> Processing 2
```

```
#> Processing 1
#> [1] 9 4 1
```

Los operadores de funciones están estrechamente relacionados con las fábricas de funciones; de hecho, son solo una fábrica de funciones que toma una función como entrada. Al igual que las fábricas, no hay nada que no puedas hacer sin ellas, pero a menudo te permiten eliminar la complejidad para que tu código sea más legible y reutilizable.

Los operadores de función suelen estar emparejados con funcionales. Si está utilizando un bucle for, rara vez hay una razón para usar un operador de función, ya que hará que su código sea más complejo con poca ganancia.

Si está familiarizado con Python, los decoradores son solo otro nombre para los operadores de funciones.

Estructura

- La Section 11.2 le presenta dos operadores de funciones existentes extremadamente útiles y le muestra cómo usarlos para resolver problemas reales.
- La Section 11.3 funciona a través de un problema susceptible de solución con operadores de función: descargar muchas páginas web.

Requisitos previos

Los operadores de funciones son un tipo de fábrica de funciones, así que asegúrese de estar familiarizado al menos con la Section 6.2 antes de continuar.

Usaremos purrr para un par de funciones que aprendiste en el Chapter 9, y algunos operadores de funciones que aprenderás a continuación. También

usaremos el paquete memoise (Wickham et al. 2018) para el operador `memoise()`.

```
library(purrr)
library(memoise)
```

11.2. Operadores de funciones existentes

Hay dos operadores de funciones muy útiles que lo ayudarán a resolver problemas recurrentes comunes y le darán una idea de lo que pueden hacer los operadores de funciones: `purrr::safely()` y `memoise::memoise()`.

11.2.1. Captura de errores con `purrr::safely()`

Una ventaja de los bucles for es que si una de las iteraciones falla, aún puede acceder a todos los resultados hasta la falla:

```
x <- list(
  c(0.512, 0.165, 0.717),
  c(0.064, 0.781, 0.427),
  c(0.890, 0.785, 0.495),
  "oops"
)

out <- rep(NA_real_, length(x))
for (i in seq_along(x)) {
  out[[i]] <- sum(x[[i]])
}
#> Error in sum(x[[i]]): invalid 'type' (character) of argument
out
#> [1] 1.39 1.27 2.17   NA
```

Si hace lo mismo con un funcional, no obtiene ningún resultado, lo que dificulta descubrir dónde radica el problema:

```
map_dbl(x, sum)
#> Error in `map_dbl()`:
#>   In index: 4.
#> Caused by error:
#> ! invalid 'type' (character) of argument
```

`purrr::safely()` proporciona una herramienta para ayudar con este problema. `safely()` es un operador de función que transforma una función para convertir errores en datos. (Puede aprender la idea básica que hace que funcione en la Section 8.6.2.) Comencemos echándole un vistazo fuera de `map_dbl()`:

```
safe_sum <- safely(sum)
safe_sum
#> function (...)
#> capture_error(.f(...), otherwise, quiet)
#> <bytecode: 0x55ad01418c40>
#> <environment: 0x55ad01418770>
```

Como todos los operadores de funciones, `safely()` toma una función y devuelve una función envuelta a la que podemos llamar como de costumbre:

```
str(safe_sum(x[[1]]))
#> List of 2
#>  $ result: num 1.39
#>  $ error : NULL
str(safe_sum(x[[4]]))
#> List of 2
#>  $ result: NULL
```

```
#>  $ error :List of 2
#>   ..$ message: chr "invalid 'type' (character) of argument"
#>   ..$ call   : language .Primitive("sum")(..., na.rm = na.rm)
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Puedes ver que una función transformada por `safely()` siempre devuelve una lista con dos elementos, `result` y `error`. Si la función se ejecuta correctamente, `error` es `NULL` y `result` contiene el resultado; si la función falla, `result` es `NULL` y `error` contiene el error.

Ahora usemos `safely()` con un funcional:

```
out <- map(x, safely(sum))
str(out)
#> List of 4
#>  $ :List of 2
#>   ..$ result: num 1.39
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 1.27
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 2.17
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: NULL
#>   ..$ error :List of 2
#>   .. ..$ message: chr "invalid 'type' (character) of argument"
#>   .. ..$ call   : language .Primitive("sum")(..., na.rm = na.rm)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condit"..
```

La salida tiene una forma un poco inconveniente, ya que tenemos cuatro listas, cada una de las cuales es una lista que contiene el `resultado` y el

`error`. Podemos hacer que la salida sea más fácil de usar girándola "al revés" con `purrr::transpose()`, de modo que obtengamos una lista de `resultados` y una lista de `errores`:

```
out <- transpose(map(x, safely(sum)))
str(out)
#> List of 2
#>  $ result:List of 4
#>   ..$ : num 1.39
#>   ..$ : num 1.27
#>   ..$ : num 2.17
#>   ..$ : NULL
#>  $ error :List of 4
#>   ..$ : NULL
#>   ..$ : NULL
#>   ..$ : NULL
#>   ..$ :List of 2
#>   .. ..$ message: chr "invalid 'type' (character) of argument"
#>   .. ..$ call   : language .Primitive("sum")(..., na.rm = na.rm)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condit"..
```

Ahora podemos encontrar fácilmente los resultados que funcionaron o las entradas que fallaron:

```
ok <- map_lgl(out$error, is.null)
ok
#> [1]  TRUE  TRUE  TRUE FALSE

x[!ok]
#> [[1]]
#> [1] "oops"

out$result[ok]
```

```
#> [[1]]
#> [1] 1.39
#>
#> [[2]]
#> [1] 1.27
#>
#> [[3]]
#> [1] 2.17
```

Puedes usar esta misma técnica en muchas situaciones diferentes. Por ejemplo, imagine que está ajustando un modelo lineal generalizado (GLM) a una lista de data frames. Los GLM a veces pueden fallar debido a problemas de optimización, pero aún desea poder intentar ajustar todos los modelos y luego mirar hacia atrás a los que fallaron:

```
fit_model <- function(df) {
  glm(y ~ x1 + x2 * x3, data = df)
}

models <- transpose(map(datasets, safely(fit_model)))
ok <- map_lgl(models$error, is.null)

# ¿Qué datos no lograron converger?
datasets[!ok]

# ¿Qué modelos tuvieron éxito?
models[ok]
```

Creo que este es un gran ejemplo del poder de combinar funcionales y operadores de funciones: `safely()` te permite expresar de manera sucinta lo que necesitas para resolver un problema común de análisis de datos.

purrr viene con otros tres operadores de función en una línea similar:

- `possibly()`: devuelve un valor predeterminado cuando hay un error. No proporciona ninguna forma de saber si ocurrió un error o no, por lo que es mejor reservarlo para los casos en los que hay algún valor centinela obvio (como `NA`).
- `quietly()`: convierte la salida, los mensajes y los efectos secundarios de advertencia en componentes de `salida`, `mensaje` y `advertencia` de la salida.
- `auto_browse()`: ejecuta automáticamente `browser()` dentro de la función cuando hay un error.

Consulte su documentación para obtener más detalles.

11.2.2. Almacenamiento en caché de cálculos con `memoise::memoise()`

Otro operador de función útil es `memoise::memoise()`. **Memoriza** una función, lo que significa que la función recordará las entradas anteriores y devolverá los resultados almacenados en caché. La memorización es un ejemplo de la compensación clásica de las ciencias de la computación entre memoria y velocidad. Una función memorizada puede ejecutarse mucho más rápido, pero debido a que almacena todas las entradas y salidas anteriores, utiliza más memoria.

Exploremos esta idea con una función de juguete que simula una operación costosa:

```
slow_function <- function(x) {
  Sys.sleep(1)
  x * 10 * runif(1)
}
system.time(print(slow_function(1)))
#> [1] 0.808
```

```
#>    user  system elapsed
#>       0       0       1

system.time(print(slow_function(1)))
#> [1] 8.34
#>    user  system elapsed
#>   0.002   0.000   1.003
```

Cuando memorizamos esta función, es lenta cuando la llamamos con nuevos argumentos. Pero cuando lo llamamos con argumentos de que se ve antes, es instantáneo: recupera el valor anterior del cómputo.

```
fast_function <- memoise::memoise(slow_function)
system.time(print(fast_function(1)))
#> [1] 6.01
#>    user  system elapsed
#>   0.001   0.000   1.002

system.time(print(fast_function(1)))
#> [1] 6.01
#>    user  system elapsed
#>   0.012   0.000   0.013
```

Un uso relativamente realista de la memorización es calcular la serie de Fibonacci. La serie de Fibonacci se define recursivamente: los dos primeros valores se definen por convención, $f(0) = 0$, $f(1) = 1$, y luego $f(n) = f(n-1) + f(n-2)$ (para cualquier entero positivo). Una versión ingenua es lenta porque, por ejemplo, `fib(10)` calcula `fib(9)` y `fib(8)`, y `fib(9)` calcula `fib(8)` y `fib(7) )`, y así sucesivamente.

```
fib <- function(n) {
  if (n < 2) return(n)
  fib(n - 2) + fib(n - 1)
```

```
}
system.time(fib(23))
#>    user  system elapsed
#>    0.03    0.00    0.03
system.time(fib(24))
#>    user  system elapsed
#>   0.046   0.000   0.047
```

Memorizar `fib()` hace que la implementación sea mucho más rápida porque cada valor se calcula solo una vez:

```
fib2 <- memoise::memoise(function(n) {
  if (n < 2) return(n)
  fib2(n - 2) + fib2(n - 1)
})
system.time(fib2(23))
#>    user  system elapsed
#>   0.006   0.000   0.006
```

Y las llamadas futuras pueden basarse en cálculos anteriores:

```
system.time(fib2(24))
#>    user  system elapsed
#>   0.001   0.000   0.001
```

Este es un ejemplo de **programación dinámica**, donde un problema complejo se puede dividir en muchos subproblemas superpuestos, y recordar los resultados de un subproblema mejora considerablemente el rendimiento.

Piense cuidadosamente antes de memorizar una función. Si la función no es **pura**, es decir, la salida no depende solo de la entrada, obtendrá resultados engañosos y confusos. Creé un error sutil en las herramientas de desarrollo porque memoricé los resultados de `available.packages()`,

que es bastante lento porque tiene que descargar un archivo grande de CRAN. Los paquetes disponibles no cambian con tanta frecuencia, pero si tiene un proceso R que se ha estado ejecutando durante algunos días, los cambios pueden volverse importantes y, dado que el problema solo surgió en los procesos R de ejecución prolongada, el error fue muy doloroso para encontrar.

11.2.3. Ejercicios

1. Base R proporciona un operador de función en forma de `Vectorize()`. ¿Qué hace? ¿Cuándo podría usarlo?

2. Lee el código fuente de `posiblemente()`. ¿Como funciona?

3. Lee el código fuente de `safely()`. ¿Como funciona?

11.3. Estudio de caso: Creación de sus propios operadores de función

`memoise()` y `safely()` son muy útiles pero también bastante complejos. En este caso de estudio, aprenderá cómo crear sus propios operadores de función más simples. Imagine que tiene un vector con nombre de URL y desea descargar cada uno en el disco. Eso es bastante simple con `walk2()` y `file.download()`:

```
urls <- c(
  "adv-r" = "https://adv-r.hadley.nz",
  "r4ds" = "http://r4ds.had.co.nz/"
  # y muchos más
)
path <- paste0(tempdir(), names(urls), ".html")
```

```
walk2(urls, path, download.file, quiet = TRUE)
```

Este enfoque está bien para un puñado de URL, pero a medida que el vector se alarga, es posible que desee agregar un par de funciones más:

- Agregue un pequeño retraso entre cada solicitud para evitar martillar el servidor.
- Mostrar un `.` cada pocas URL para que sepamos que la función sigue funcionando.

Es relativamente fácil agregar estas funciones adicionales si usamos un bucle for:

```
for(i in seq_along(urls)) {
  Sys.sleep(0.1)
  if (i %% 10 == 0) cat(".")
  download.file(urls[[i]], paths[[i]])
}
```

Creo que este ciclo for es subóptimo porque intercala diferentes preocupaciones: pausar, mostrar el progreso y descargar. Esto hace que el código sea más difícil de leer y dificulta la reutilización de los componentes en situaciones nuevas. En cambio, veamos si podemos usar operadores de función para extraer la pausa y mostrar el progreso y hacerlos reutilizables.

Primero, escribamos un operador de función que agregue un pequeño retraso. Voy a llamarlo `delay_by()` por razones que serán más claras en breve, y tiene dos argumentos: la función para envolver y la cantidad de retraso para agregar. La implementación real es bastante simple. El truco principal es forzar la evaluación de todos los argumentos como se describe en la Section 10.2.5, porque los operadores de función son un tipo especial de fábrica de funciones:

```
delay_by <- function(f, amount) {
  force(f)
  force(amount)

  function(...) {
    Sys.sleep(amount)
    f(...)
  }
}
system.time(runif(100))
#>    user  system elapsed
#>   0.001   0.000   0.000
system.time(delay_by(runif, 0.1)(100))
#>    user  system elapsed
#>     0.0     0.0     0.1
```

Y podemos usarlo con el `walk2()` original:

```
walk2(urls, path, delay_by(download.file, 0.1), quiet = TRUE)
```

Crear una función para mostrar el punto ocasional es un poco más difícil, porque ya no podemos confiar en el índice del bucle. Podríamos pasar el índice como otro argumento, pero eso rompe la encapsulación: una preocupación de la función de progreso ahora se convierte en un problema que el contenedor de nivel superior debe manejar. En su lugar, usaremos otro truco de fábrica de funciones (de la Section 10.2.4), para que el contenedor de progreso pueda administrar su propio contador interno:

```
dot_every <- function(f, n) {
  force(f)
  force(n)
```

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

Ahora podemos expresar nuestro bucle for original como:

```
walk2(
  urls, path,
  dot_every(delay_by(download.file, 0.1), 10),
  quiet = TRUE
)
```

Esto está empezando a ser un poco difícil de leer porque estamos componiendo muchas llamadas a funciones y los argumentos se están dispersando. Una forma de resolver eso es usar la tubería:

```
walk2(
  urls, path,
  download.file |> dot_every(10) |> delay_by(0.1),
  quiet = TRUE
)
```

La canalización funciona bien aquí porque elegí cuidadosamente los nombres de las funciones para generar una oración (casi) legible: tome `download.file` luego (agregue) un punto cada 10 iteraciones, luego

retrase 0.1s. Cuanto más claramente pueda expresar la intención de su código a través de nombres de funciones, más fácilmente otros (¡incluido usted en el futuro!) podrán leer y comprender el código.

11.3.1. Ejercicios

1. Sopesar los pros y los contras de `download.file |> dot_every(10) |> delay_by(0.1)` versus `download.file |> delay_by(0.1) |> dot_every(10)`.
2. ¿Deberías memorizar `download.file()`? ¿Por qué o por qué no?
3. Cree un operador de función que informe cada vez que se crea o elimina un archivo en el directorio de trabajo, usando `dir()` y `setdiff()`. ¿Qué otros efectos de funciones globales le gustaría rastrear?
4. Escriba un operador de función que registre una marca de tiempo y un mensaje en un archivo cada vez que se ejecute una función.
5. Modifique `delay_by()` para que, en lugar de retrasar una cantidad de tiempo fija, asegure que haya transcurrido una cierta cantidad de tiempo desde la última vez que se llamó a la función. Es decir, si llamó a `g <- delay_by(1, f); g(); Sys.sleep(2); g()` no debería haber un retraso adicional.

Part III.

Programación orientada a objetos

Introdución

En los cinco capítulos siguientes, aprenderá sobre la **programación orientada a objetos** (POO). POO es un poco más desafiante en R que en otros lenguajes porque:

- Hay múltiples sistemas de POO para elegir. En este libro, me concentraré en los tres que considero más importantes: **S3**, **R6** y **S4**. S3 y S4 son proporcionados por la base R. R6 es proporcionado por el paquete R6 y es similar a las clases de referencia, o **RC** para abreviar, desde la base R.

- Hay desacuerdo sobre la importancia relativa de los sistemas de POO. Creo que S3 es el más importante, seguido de R6, luego S4. Otros creen que S4 es el más importante, seguido de RC, y que S3 debe evitarse. Esto significa que diferentes comunidades R usan diferentes sistemas.

- S3 y S4 utilizan la función POO genérica, que es bastante diferente de la programación orientada a objetos encapsulada utilizada por la mayoría de los lenguajes populares en la actualidad[1]. Volveremos precisamente a lo que significan esos términos en breve, pero básicamente, aunque las ideas subyacentes de POO son las mismas en todos los idiomas, sus expresiones son bastante diferentes. Esto significa que no puede transferir inmediatamente sus habilidades de POO existentes a R.

[1]La excepción es Julia, que también usa la función genérica de POO. En comparación con R, la implementación de Julia está completamente desarrollada y tiene un rendimiento extremo.

En general, en R, la programación funcional es mucho más importante que la programación orientada a objetos, porque normalmente resuelve problemas complejos descomponiéndolos en funciones simples, no en objetos simples. Sin embargo, hay razones importantes para aprender cada uno de los tres sistemas:

- S3 permite que sus funciones devuelvan resultados enriquecidos con una pantalla fácil de usar y componentes internos fáciles de usar para el programador. S3 se usa en toda la base R, por lo que es importante dominarlo si desea extender las funciones base R para trabajar con nuevos tipos de entrada.

- R6 proporciona una forma estandarizada de escapar de la semántica de copia al modificar de R. Esto es particularmente importante si desea modelar objetos que existen independientemente de R. Hoy en día, una necesidad común para R6 es modelar datos que provienen de una API web y dónde los cambios provienen de dentro o fuera de R.

- S4 es un sistema riguroso que lo obliga a pensar cuidadosamente sobre el diseño del programa. Es particularmente adecuado para construir grandes sistemas que evolucionan con el tiempo y recibirá contribuciones de muchos programadores. Es por eso que lo utiliza el proyecto Bioconductor, por lo que otra razón para aprender S4 es equiparlo para contribuir a ese proyecto.

El objetivo de este breve capítulo introductorio es brindarle un vocabulario importante y algunas herramientas para identificar los sistemas de POO en la naturaleza. Luego, los siguientes capítulos se sumergen en los detalles de los sistemas de POO de R:

1. Chapter 12 detalla los tipos básicos que forman la base subyacente a todos los demás sistemas OO.

2. Chapter 13 presenta S3, el sistema OO más simple y más utilizado.

3. Chapter 14 analiza R6, un sistema OO encapsulado creado sobre entornos.

4. Chapter 15 introduce S4, que es similar a S3 pero más formal y más estricto.

5. Chapter 16 compara estos tres sistemas OO principales. Al comprender las ventajas y desventajas de cada sistema, puede apreciar cuándo usar uno u otro.

Este libro se centra en la mecánica de la programación orientada a objetos, no en su uso efectivo, y puede ser un desafío comprenderlo completamente si no ha realizado antes programación orientada a objetos. Quizás se pregunte por qué elegí no proporcionar una cobertura útil más inmediata. Me he centrado en la mecánica aquí porque necesitan estar bien descritas en alguna parte (escribir estos capítulos requirió una cantidad considerable de lectura, exploración y síntesis de mi parte), y usar OOP de manera efectiva es lo suficientemente complejo como para requerir un tratamiento del tamaño de un libro; simplemente no hay suficiente espacio en *R Avanzado* para cubrirlo con la profundidad requerida.

Sistemas de POO

Diferentes personas usan los términos de programación orientada a objetos de diferentes maneras, por lo que esta sección proporciona una descripción general rápida del vocabulario importante. Las explicaciones están necesariamente comprimidas, pero volveremos a estas ideas varias veces.

La razón principal para usar POO es **polimorfismo** (literalmente: muchas formas). El polimorfismo significa que un desarrollador puede considerar la interfaz de una función por separado de su implementación, lo que hace posible usar la misma forma de función para diferentes tipos de entrada. Esto está estrechamente relacionado con la idea de **encapsulación**: el

usuario no necesita preocuparse por los detalles de un objeto porque están encapsulados detrás de una interfaz estándar.

Para ser concretos, el polimorfismo es lo que permite que `summary()` produzca diferentes salidas para variables numéricas y factoriales:

```
diamonds <- ggplot2::diamonds

summary(diamonds$carat)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    0.20    0.40    0.70    0.80    1.04    5.01

summary(diamonds$cut)
#>      Fair      Good Very Good   Premium     Ideal
#>      1610      4906     12082     13791     21551
```

Podrías imaginar `summary()` que contiene una serie de declaraciones if-else, pero eso significaría que solo el autor original podría agregar nuevas implementaciones. Un sistema OOP hace posible que cualquier desarrollador amplíe la interfaz con implementaciones para nuevos tipos de entrada.

Para ser más precisos, los sistemas OO llaman al tipo de un objeto su **clase**, y una implementación para una clase específica se llama **método**. En términos generales, una clase define lo que un objeto *es* y los métodos describen lo que ese objeto puede *hacer*. La clase define los **campos**, los datos que posee cada instancia de esa clase. Las clases están organizadas en una jerarquía de modo que si no existe un método para una clase, se usa el método de su padre y se dice que el hijo **hereda** el comportamiento. Por ejemplo, en R, un factor ordenado hereda de un factor regular y un modelo lineal generalizado hereda de un modelo lineal. El proceso de encontrar el método correcto dada una clase se llama **despacho de métodos**.

Hay dos paradigmas principales de programación orientada a objetos que difieren en cómo se relacionan los métodos y las clases. En este libro,

tomaremos prestada la terminología de *Extending R* (Chambers 2016) y llamaremos a estos paradigmas encapsulados y funcionales:

- En la programación orientada a objetos **encapsulada**, los métodos pertenecen a objetos o clases, y las llamadas a métodos normalmente se ven como `object.method(arg1, arg2)`. Esto se denomina encapsulado porque el objeto encapsula tanto los datos (con campos) como el comportamiento (con métodos), y es el paradigma que se encuentra en los lenguajes más populares.
- En la programación orientada a objetos **funcional**, los métodos pertenecen a funciones **genéricas** y las llamadas a métodos se parecen a las llamadas a funciones ordinarias: `generic(object, arg2, arg3)`. Esto se llama funcional porque desde el exterior parece una llamada de función regular, e internamente los componentes también son funciones.

Con esta terminología en la mano, ahora podemos hablar precisamente de los diferentes sistemas OO disponibles en R.

POO en R

Base R proporciona tres sistemas OOP: S3, S4 y clases de referencia (RC):

- **S3** es el primer sistema de POO de R y se describe en *Modelos estadísticos en S* (Chambers and Hastie 1992). S3 es una implementación informal de POO funcional y se basa en convenciones comunes en lugar de garantías inquebrantables. Esto hace que sea fácil comenzar, proporcionando una forma económica de resolver muchos problemas simples.
- **S4** es una reescritura formal y rigurosa de S3 y se introdujo en *Programación con datos* (Chambers 1998). Requiere más trabajo

inicial que S3, pero a cambio ofrece más garantías y una mayor encapsulación. S4 se implementa en el paquete base de **métodos**, que siempre se instala con R.

(Quizás se pregunte si existen S1 y S2. No lo hacen: S3 y S4 fueron nombrados de acuerdo con las versiones de S que acompañaban. Las dos primeras versiones de S no tenían ningún marco de POO.)

- **RC** implementa OO encapsulado. Los objetos RC son un tipo especial de objetos S4 que también son **mutables**, es decir, en lugar de usar la semántica habitual de copiar al modificar de R, se pueden modificar en su lugar. Esto los hace más difíciles de razonar, pero les permite resolver problemas que son difíciles de resolver en el estilo OOP funcional de S3 y S4.

Los paquetes CRAN proporcionan otros sistemas de POO:

- **R6** (Chang 2017) implementa OOP encapsulado como RC, pero resuelve algunos problemas importantes. En este libro, aprenderá sobre R6 en lugar de RC, por las razones descritas en la Section 14.5.

- **R.oo** (Bengtsson 2003) proporciona algo de formalismo además de S3 y hace posible tener objetos mutables de S3.

- **proto** (Grothendieck, Kates, and Petzoldt 2016) implementa otro estilo de programación orientada a objetos basado en la idea de **prototipos**, que desdibujan las distinciones entre clases e instancias de clases (objetos). Me enamoré brevemente de la programación basada en prototipos (Wickham 2011) y la usé en ggplot2, pero ahora creo que es mejor seguir con los formularios estándar.

Aparte del R6, que es ampliamente utilizado, estos sistemas son principalmente de interés teórico. Tienen sus puntos fuertes, pero pocos usuarios de R los conocen y los entienden, por lo que es difícil que otros los lean y contribuyan a su código.

sloop

Antes de continuar, quiero presentar el paquete sloop:

```
library(sloop)
```

El paquete sloop (piense en "navegar los mares de OOP") proporciona una serie de ayudantes que completan las piezas que faltan en la base R. El primero de ellos es `sloop::otype()`. Hace que sea fácil descifrar el sistema OOP utilizado por un objeto capturado de forma salvaje:

```
otype(1:10)
#> [1] "base"

otype(mtcars)
#> [1] "S3"

mle_obj <- stats4::mle(function(x = 1) (x - 2) ^ 2)
otype(mle_obj)
#> [1] "S4"
```

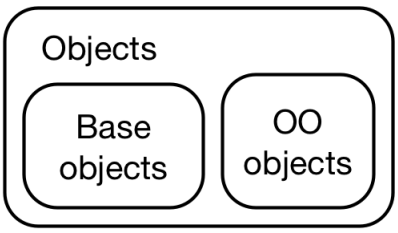
Utilice esta función para averiguar qué capítulo leer para entender cómo trabajar con un objeto existente.

12. Tipos básicos

12.1. Introducción

Para hablar de objetos y programación orientada a objetos en R, primero debemos aclarar una confusión fundamental sobre dos usos de la palabra "objeto". Hasta ahora en este libro, hemos usado la palabra en el sentido general captado por la concisa cita de John Chambers: "Todo lo que existe en R es un objeto". Sin embargo, aunque todo *es* un objeto, no todo está orientado a objetos. Esta confusión surge porque los objetos base provienen de S y se desarrollaron antes de que nadie pensara que S podría necesitar un sistema OOP. Las herramientas y la nomenclatura evolucionaron orgánicamente durante muchos años sin un solo principio rector.

La mayoría de las veces, la distinción entre objetos y objetos orientados a objetos no es importante. Pero aquí tenemos que entrar en detalles esenciales, así que usaremos los términos **objetos base** y **objetos OO** para distinguirlos.



Estructura

- La Section 12.2 le muestra cómo identificar objetos base y OO.
- La Section 12.3 proporciona un conjunto completo de los tipos base utilizados para construir todos los objetos.

12.2. Base versus objetos OO

Para saber la diferencia entre un objeto base y OO, usa `is.object()` o `sloop::otype()`:

```
# Un objeto básico:
is.object(1:10)
#> [1] FALSE
sloop::otype(1:10)
#> [1] "base"

# Un objeto OO
is.object(mtcars)
#> [1] TRUE
sloop::otype(mtcars)
#> [1] "S3"
```

Técnicamente, la diferencia entre los objetos base y OO es que los objetos OO tienen un atributo de "clase":

```
attr(1:10, "class")
#> NULL

attr(mtcars, "class")
#> [1] "data.frame"
```

Puede que ya estés familiarizado con la función `class()`. Es seguro aplicar esta función a objetos S3 y S4, pero devuelve resultados engañosos cuando se aplica a objetos base. Es más seguro usar `sloop::s3_class()`, que devuelve la clase implícita que los sistemas S3 y S4 usarán para seleccionar métodos. Aprenderá más sobre `s3_class()` en la Section 13.7.1.

```
x <- matrix(1:4, nrow = 2)
class(x)
#> [1] "matrix" "array"
sloop::s3_class(x)
#> [1] "matrix"  "integer" "numeric"
```

12.3. Tipos básicos

Mientras que solo los objetos OO tienen un atributo de clase, cada objeto tiene un **tipo base**:

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

typeof(mtcars)
#> [1] "list"
```

Los tipos base no forman un sistema OOP porque las funciones que se comportan de manera diferente para diferentes tipos base se escriben principalmente en código C que usa instrucciones de cambio. Esto significa que solo R-core puede crear nuevos tipos, y crear un nuevo tipo es mucho trabajo porque cada declaración de cambio debe modificarse para manejar un nuevo caso. Como consecuencia, rara vez se agregan nuevos tipos base. El cambio más reciente, en 2011, agregó dos tipos exóticos que nunca se

ven en R, pero que son necesarios para diagnosticar problemas de memoria. Antes de eso, el último tipo agregado fue un tipo base especial para objetos S4 agregado en 2005.

En total, hay 25 tipos de base diferentes. Se enumeran a continuación, agrupados libremente según el lugar en el que se analicen en este libro. Estos tipos son los más importantes en el código C, por lo que a menudo los verá llamados por sus nombres de tipo C. Los he incluido entre paréntesis.

- Vectores, Chapter 3, incluye tipos `NULL` (`NILSXP`), `logical` (`LGLSXP`), `integer` (`INTSXP`), `double` (`REALSXP`), `complex` (`CPLXSXP`), `character` (`STRSXP`), `list` (`VECSXP`), y `raw` (`RAWSXP`).

  ```
  typeof(NULL)
  #> [1] "NULL"
  typeof(1L)
  #> [1] "integer"
  typeof(1i)
  #> [1] "complex"
  ```

- Las funciones, Chapter 6, incluyen los tipos `cierre` (funciones regulares de R, `CLOSXP`), `especiales` (funciones internas, `SPECIALSXP`) e `incorporadas` (funciones primitivas, `BUILTINSXP`).

  ```
  typeof(mean)
  #> [1] "closure"
  typeof(`[`)
  #> [1] "special"
  typeof(sum)
  #> [1] "builtin"
  ```

  Las funciones internas y primitivas se describen en la Section 6.2.2.

- Entornos, Chapter 7, tienen tipo `entorno` (`ENVSXP`).

```
typeof(globalenv())
#> [1] "environment"
```

- El tipo `S4` (`S4SXP`), Chapter 15, se usa para las clases de S4 que no heredan de un tipo base existente.

```
mle_obj <- stats4::mle(function(x = 1) (x - 2) ^ 2)
typeof(mle_obj)
#> [1] "S4"
```

- Los componentes del lenguaje, Chapter 18), incluyen `símbolo` (también conocido como nombre, `SYMSXP`), `idioma` (generalmente llamadas llamadas, `LANGSXP`) y `pairlist` (usado para argumentos de función, `LISTSXP`) tipos.

```
typeof(quote(a))
#> [1] "symbol"
typeof(quote(a + 1))
#> [1] "language"
typeof(formals(mean))
#> [1] "pairlist"
```

  `expression` (`EXPRSXP`) es un tipo de propósito especial que solo es devuelto por `parse()` y `expression()`. Las expresiones generalmente no son necesarias en el código de usuario.

- Los tipos restantes son esotéricos y rara vez se ven en R. Son importantes principalmente para el código C: `externalptr` (`EXTPTRSXP`), `weakref` (`WEAKREFSXP`), `bytecode` (`BCODESXP`), `promise` (`PROMSXP`), `...` (`DOTSXP`), y `any` (`ANYSXP`).

Es posible que hayas oído hablar de `mode()` y `storage.mode()`. No utilice estas funciones: solo existen para proporcionar nombres de tipo que sean compatibles con S.

12.3.1. Tipo numérico

Tenga cuidado al hablar del tipo numérico, porque R usa "numérico" para referirse a tres cosas ligeramente diferentes:

1. En algunos lugares, numérico se usa como un alias para el tipo doble. Por ejemplo, `as.numeric()` es idéntico a `as.double()`, y `numeric()` es idéntico a `double()`.

   (R también usa ocasionalmente real en lugar de doble; `NA_real_` es el único lugar donde es probable que encuentres esto en la práctica.)

2. En los sistemas S3 y S4, numérico se usa como una forma abreviada de tipo entero o doble, y se usa cuando se seleccionan métodos:

   ```
   sloop::s3_class(1)
   #> [1] "double"  "numeric"
   sloop::s3_class(1L)
   #> [1] "integer" "numeric"
   ```

3. `is.numeric()` pruebas para objetos que se *comportan* como números. Por ejemplo, los factores tienen el tipo "entero" pero no se comportan como números (es decir, no tiene sentido tomar la media del factor).

   ```
   typeof(factor("x"))
   #> [1] "integer"
   is.numeric(factor("x"))
   #> [1] FALSE
   ```

En este libro, siempre uso numérico para indicar un objeto de tipo entero o doble.

13. S3

13.1. Introducción

S3 es el primer y más simple sistema OO de R. S3 es informal y ad hoc, pero hay cierta elegancia en su minimalismo: no se le puede quitar ninguna parte y seguir teniendo un sistema OO útil. Por estas razones, debe usarlo, a menos que tenga una razón convincente para hacerlo de otra manera. S3 es el único sistema OO utilizado en los paquetes base y stats, y es el sistema más utilizado en los paquetes CRAN.

S3 es muy flexible, lo que significa que te permite hacer cosas que son bastante desaconsejables. Si viene de un entorno estricto como Java, esto parecerá bastante aterrador, pero le da a los programadores de R una gran libertad. Puede ser muy difícil evitar que las personas hagan algo que usted no quiere que hagan, pero sus usuarios nunca se detendrán porque hay algo que aún no ha implementado. Dado que S3 tiene pocas restricciones integradas, la clave para su uso exitoso es aplicar las restricciones usted mismo. Por lo tanto, este capítulo le enseñará las convenciones que debe seguir (casi) siempre.

El objetivo de este capítulo es mostrarle cómo funciona el sistema S3, no cómo usarlo de manera efectiva para crear nuevas clases y genéricos. Recomiendo combinar el conocimiento teórico de este capítulo con el conocimiento práctico codificado en el paquete vctrs.

Estructura

- La Section 13.2 brinda una descripción general rápida de todos los componentes principales de S3: clases, genéricos y métodos. También aprenderá sobre `sloop::s3_dispatch()`, que usaremos a lo largo del capítulo para explorar cómo funciona S3.
- La Section 13.3 entra en los detalles de la creación de una nueva clase S3, incluidas las tres funciones que deberían acompañar a la mayoría de las clases: un constructor, un ayudante y un validador.
- La Section 13.4 describe cómo funcionan los métodos y genéricos de S3, incluidos los aspectos básicos del envío de métodos.
- La Section 13.5 analiza los cuatro estilos principales de los objetos de S3: vector, registro, marco de datos y escalar.
- La Section 13.6 demuestra cómo funciona la herencia en S3 y le muestra lo que necesita para hacer que una clase sea "subclasificable".
- La Section 13.7 concluye el capítulo con una discusión de los detalles más finos del envío de métodos, incluidos los tipos base, los genéricos internos, los genéricos de grupo y el envío doble.

Requisitos previos

Las clases de S3 se implementan mediante atributos, así que asegúrese de estar familiarizado con los detalles descritos en la Section 3.3. Usaremos vectores S3 base existentes para ejemplos y exploración, así que asegúrese de estar familiarizado con las clases factor, Date, difftime, POSIXct y POSIXlt descritas en la Section 3.4.

Usaremos el paquete sloop para sus ayudantes interactivos.

```
library(sloop)
```

13.2. Lo esencial

Un objeto S3 es un tipo base con al menos un atributo de "clase" (se pueden usar otros atributos para almacenar otros datos). Por ejemplo, tome el factor. Su tipo base es el vector entero, tiene un atributo `clase` de "factor", y un atributo `niveles` que almacena los niveles posibles:

```
f <- factor(c("a", "b", "c"))

typeof(f)
#> [1] "integer"
attributes(f)
#> $levels
#> [1] "a" "b" "c"
#>
#> $class
#> [1] "factor"
```

Puede obtener el tipo base subyacente al `unclass()`, lo que elimina el atributo de clase, lo que hace que pierda su comportamiento especial:

```
unclass(f)
#> [1] 1 2 3
#> attr(,"levels")
#> [1] "a" "b" "c"
```

Un objeto de S3 se comporta de manera diferente a su tipo base subyacente cada vez que se pasa a un **genérico** (abreviatura de función genérica). La forma más fácil de saber si una función es genérica es usar `sloop::ftype()` y buscar "genérica" en la salida:

```
ftype(print)
#> [1] "S3"      "generic"
ftype(str)
#> [1] "S3"      "generic"
ftype(unclass)
#> [1] "primitive"
```

Una función genérica define una interfaz, que utiliza una implementación diferente según la clase de un argumento (casi siempre el primer argumento). Muchas funciones básicas de R son genéricas, incluida la importante `print()`:

```
print(f)
#> [1] a b c
#> Levels: a b c

# la eliminación de clase vuelve al comportamiento de entero
print(unclass(f))
#> [1] 1 2 3
#> attr(,"levels")
#> [1] "a" "b" "c"
```

Tenga en cuenta que `str()` es genérico, y algunas clases de S3 usan ese genérico para ocultar los detalles internos. Por ejemplo, la clase `POSIXlt` que se usa para representar datos de fecha y hora en realidad está construida encima de una lista, un hecho que está oculto por su método `str()`:

```
time <- strptime(c("2017-01-01", "2020-05-04 03:21"), "%Y-%m-%d")
str(time)
#>  POSIXlt[1:2], format: "2017-01-01" "2020-05-04"

str(unclass(time))
```

```
#> List of 11
#>  $ sec   : num [1:2] 0 0
#>  $ min   : int [1:2] 0 0
#>  $ hour  : int [1:2] 0 0
#>  $ mday  : int [1:2] 1 4
#>  $ mon   : int [1:2] 0 4
#>  $ year  : int [1:2] 117 120
#>  $ wday  : int [1:2] 0 1
#>  $ yday  : int [1:2] 0 124
#>  $ isdst : int [1:2] 0 0
#>  $ zone  : chr [1:2] "UTC" "UTC"
#>  $ gmtoff: int [1:2] 0 0
#>  - attr(*, "tzone")= chr "UTC"
#>  - attr(*, "balanced")= logi TRUE
```

El genérico es un intermediario: su trabajo es definir la interfaz (es decir, los argumentos) y luego encontrar la implementación correcta para el trabajo. La implementación para una clase específica se denomina **método**, y el genérico encuentra ese método realizando **despacho de métodos**.

Puede usar `sloop::s3_dispatch()` para ver el proceso de envío del método:

```
s3_dispatch(print(f))
#> => print.factor
#>  * print.default
```

Volveremos a los detalles del envío en la Section 13.4.1, por ahora tenga en cuenta que los métodos S3 son funciones con un esquema de nombres especial, `generic.class()`. Por ejemplo, el método `factor` para el genérico `print()` se llama `print.factor()`. Nunca debe llamar al método directamente, sino confiar en el genérico para encontrarlo por usted.

En general, puede identificar un método por la presencia de `.` en el nombre de la función, pero hay una serie de funciones importantes en base R que se escribieron antes de S3 y, por lo tanto, usan `.` para unir palabras. Si no está seguro, verifique con `sloop::ftype()`:

```
ftype(t.test)
#> [1] "S3"      "generic"
ftype(t.data.frame)
#> [1] "S3"     "method"
```

A diferencia de la mayoría de las funciones, no puede ver el código fuente de la mayoría de los métodos S3 [1] simplemente escribiendo sus nombres. Esto se debe a que los métodos de S3 generalmente no se exportan: viven solo dentro del paquete y no están disponibles en el entorno global. En su lugar, puede usar `sloop::s3_get_method()`, que funcionará independientemente de dónde resida el método:

```
weighted.mean.Date
#> Error in eval(expr, envir, enclos): object 'weighted.mean.Date' not found

s3_get_method(weighted.mean.Date)
#> function (x, w, ...)
#> .Date(weighted.mean(unclass(x), w, ...))
#> <bytecode: 0x559e9c3a15c0>
#> <environment: namespace:stats>
```

13.2.1. Ejercicios

1. Describe la diferencia entre `t.test()` y `t.data.frame()`. ¿Cuándo se llama cada función?

[1] Las excepciones son los métodos que se encuentran en el paquete base, como `t.data.frame`, y los métodos que ha creado.

2. Haga una lista de las funciones básicas de R que se usan comúnmente y que contienen `.` en su nombre, pero que no son métodos de S3.

3. ¿Qué hace el método `as.data.frame.data.frame()`? ¿Por qué es confuso? ¿Cómo podría evitar esta confusión en su propio código?

4. Describa la diferencia de comportamiento en estas dos llamadas.

```
set.seed(1014)
some_days <- as.Date("2017-01-31") + sample(10, 5)

mean(some_days)
#> [1] "2017-02-06"
mean(unclass(some_days))
#> [1] 17203
```

5. ¿Qué clase de objeto devuelve el siguiente código? ¿Sobre qué tipo de base está construido? ¿Qué atributos utiliza?

```
x <- ecdf(rpois(100, 10))
x
#> Empirical CDF
#> Call: ecdf(rpois(100, 10))
#>  x[1:18] =  2,  3,  4,  ..., 2e+01, 2e+01
```

6. ¿Qué clase de objeto devuelve el siguiente código? ¿Sobre qué tipo de base está construido? ¿Qué atributos utiliza?

```
x <- table(rpois(100, 5))
x
#>
#>  1  2  3  4  5  6  7  8  9 10
#>  7  5 18 14 15 15 14  4  5  3
```

13.3. Clases

Si ha realizado programación orientada a objetos en otros lenguajes, se sorprenderá al saber que S3 no tiene una definición formal de una clase: para convertir un objeto en una instancia de una clase, simplemente establezca el **atributo de clase**. Puedes hacerlo durante la creación con `structure()`, o después del hecho con `class<-()`:

```
# Crear y asignar clases en un solo paso
x <- structure(list(), class = "my_class")

# Crear, luego establecer la clase
x <- list()
class(x) <- "my_class"
```

Puede determinar la clase de un objeto S3 con `class(x)` y ver si un objeto es una instancia de una clase usando `inherits(x, "classname")`.

```
class(x)
#> [1] "my_class"
inherits(x, "my_class")
#> [1] TRUE
inherits(x, "your_class")
#> [1] FALSE
```

El nombre de la clase puede ser cualquier cadena, pero recomiendo usar solo letras y `_`. Evite `.` porque (como se mencionó anteriormente) puede confundirse con el separador `.` entre un nombre genérico y un nombre de clase. Al usar una clase en un paquete, recomiendo incluir el nombre del paquete en el nombre de la clase. Eso asegura que no chocará accidentalmente con una clase definida por otro paquete.

S3 no tiene comprobaciones de corrección, lo que significa que puede cambiar la clase de los objetos existentes:

```
# Crear un modelo lineal
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
#> [1] "lm"
print(mod)
#>
#> Call:
#> lm(formula = log(mpg) ~ log(disp), data = mtcars)
#>
#> Coefficients:
#> (Intercept)    log(disp)
#>       5.381       -0.459

# Conviértelo en una fecha (?!)
class(mod) <- "Date"

# Como era de esperar, esto no funciona muy bien
print(mod)
#> Error in as.POSIXlt(.Internal(Date2POSIXlt(x, tz)), tz = tz): 'list' object cannot be coe
```

Si ha usado otros lenguajes orientados a objetos, esto podría hacerle sentir mareado, pero en la práctica esta flexibilidad causa pocos problemas. R no evita que te dispares en el pie, pero mientras no apuntes el arma a los dedos de los pies y aprietes el gatillo, no tendrás ningún problema.

Para evitar las intersecciones de pie y bala al crear su propia clase, le recomiendo que proporcione generalmente tres funciones:

- Un **constructor** de bajo nivel, `new_myclass()`, que crea eficientemente nuevos objetos con la estructura correcta.
- Un **validador**, `validate_myclass()`, que realiza verificaciones más costosas desde el punto de vista computacional para garantizar que el objeto tenga los valores correctos.

- Un **ayudante** fácil de usar, `myclass()`, que proporciona una manera conveniente para que otros creen objetos de su clase.

No necesita un validador para clases muy simples, y puede omitir el asistente si la clase es solo para uso interno, pero siempre debe proporcionar un constructor.

13.3.1. Constructores

S3 no proporciona una definición formal de una clase, por lo que no tiene una forma integrada de garantizar que todos los objetos de una clase determinada tengan la misma estructura (es decir, el mismo tipo base y los mismos atributos con los mismos tipos). En su lugar, debe aplicar una estructura coherente mediante el uso de un **constructor**.

El constructor debe seguir tres principios:

- Se llamará `new_myclass()`.
- Tener un argumento para el objeto base y uno para cada atributo.
- Comprobar el tipo del objeto base y los tipos de cada atributo.

Ilustraré estas ideas creando constructores para las clases base[2] con las que ya está familiarizado. Para comenzar, hagamos un constructor para la clase S3 más simple: `Date`. Una fecha es simplemente un doble con un único atributo: su `clase` es `Date`. Esto lo convierte en un constructor muy simple:

[2]Las versiones recientes de R tienen constructores `.Date()`, `.difftime()`, `.POSIXct()` y `.POSIXlt()`, pero son internos, no están bien documentados y no siguen los principios que Recomiendo.

```
new_Date <- function(x = double()) {
  stopifnot(is.double(x))
  structure(x, class = "Date")
}

new_Date(c(-1, 0, 1))
#> [1] "1969-12-31" "1970-01-01" "1970-01-02"
```

El propósito de los constructores es ayudarte a ti, el desarrollador. Eso significa que puede mantenerlos simples y no necesita optimizar los mensajes de error para el consumo público. Si espera que los usuarios también creen objetos, debe crear una función de ayuda amigable, llamada `class_name()`, que describiré en breve.

Un constructor un poco más complicado es el de `difftime`, que se usa para representar diferencias de tiempo. Se basa de nuevo en un doble, pero tiene un atributo de `unidades` que debe tomar uno de un pequeño conjunto de valores:

```
new_difftime <- function(x = double(), units = "secs") {
  stopifnot(is.double(x))
  units <- match.arg(units, c("secs", "mins", "hours", "days", "weeks"))

  structure(x,
    class = "difftime",
    units = units
  )
}

new_difftime(c(1, 10, 3600), "secs")
#> Time differences in secs
#> [1]    1   10 3600
new_difftime(52, "weeks")
#> Time difference of 52 weeks
```

El constructor es una función de desarrollador: será llamado en muchos lugares por un usuario experimentado. Eso significa que está bien intercambiar un poco de seguridad a cambio de rendimiento, y debe evitar verificaciones potencialmente lentas en el constructor.

13.3.2. Validadores

Las clases más complicadas requieren controles de validez más complicados. Tome factores, por ejemplo. Un constructor solo verifica que los tipos sean correctos, lo que permite crear factores con formato incorrecto:

```
new_factor <- function(x = integer(), levels = character()) {
  stopifnot(is.integer(x))
  stopifnot(is.character(levels))

  structure(
    x,
    levels = levels,
    class = "factor"
  )
}

new_factor(1:5, "a")
#> Error in as.character.factor(x): malformed factor
new_factor(0:1, "a")
#> Error in as.character.factor(x): malformed factor
```

En lugar de sobrecargar al constructor con controles complicados, es mejor ponerlos en una función separada. Si lo hace, le permite crear nuevos objetos de forma económica cuando sabe que los valores son correctos y reutilizar fácilmente las comprobaciones en otros lugares.

```
validate_factor <- function(x) {
  values <- unclass(x)
  levels <- attr(x, "levels")

  if (!all(!is.na(values) & values > 0)) {
    stop(
      "Todos los valores `x` deben ser no faltantes y mayores que cero",
      call. = FALSE
    )
  }

  if (length(levels) < max(values)) {
    stop(
      "Debe haber al menos tantos `levels` como valores posibles en `x`",
      call. = FALSE
    )
  }

  x
}

validate_factor(new_factor(1:5, "a"))
#> Error: Debe haber al menos tantos `levels` como valores posibles en `x`
validate_factor(new_factor(0:1, "a"))
#> Error: Todos los valores `x` deben ser no faltantes y mayores que cero
```

Esta función de validación se llama principalmente por sus efectos secundarios (arrojar un error si el objeto no es válido), por lo que esperaría que devuelva su entrada principal de forma invisible (como se describe en la Section 6.7.2). Sin embargo, es útil que los métodos de validación regresen visiblemente, como veremos a continuación.

13.3.3. Ayudantes

Si desea que los usuarios construyan objetos de su clase, también debe proporcionar un método auxiliar que les haga la vida lo más fácil posible. Un ayudante siempre debe:

- Tener el mismo nombre que la clase, p. `myclass()`.
- Termine llamando al constructor y al validador, si existe.
- Cree mensajes de error cuidadosamente elaborados y adaptados a un usuario final.
- Tenga una interfaz de usuario cuidadosamente diseñada con valores predeterminados cuidadosamente seleccionados y conversiones útiles.

La última viñeta es la más complicada y es difícil dar consejos generales. Sin embargo, hay tres patrones comunes:

- A veces, todo lo que necesita hacer el ayudante es forzar sus entradas al tipo deseado. Por ejemplo, `new_difftime()` es muy estricto y viola la convención habitual de que puede usar un vector entero siempre que pueda usar un vector doble:

  ```
  new_difftime(1:10)
  #> Error in new_difftime(1:10): is.double(x) is not TRUE
  ```

  No es el trabajo del constructor ser flexible, así que aquí creamos un ayudante que solo fuerza la entrada a un doble.

  ```
  difftime <- function(x = double(), units = "secs") {
    x <- as.double(x)
    new_difftime(x, units = units)
  }

  difftime(1:10)
  ```

```
#> Time differences in secs
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

- A menudo, la representación más natural de un objeto complejo es una cadena. Por ejemplo, es muy conveniente especificar factores con un vector de caracteres. El siguiente código muestra una versión simple de `factor()`: toma un vector de caracteres y supone que los niveles deberían ser valores únicos. Esto no siempre es correcto (ya que es posible que algunos niveles no se vean en los datos), pero es un valor predeterminado útil.

```
factor <- function(x = character(), levels = unique(x)) {
  ind <- match(x, levels)
  validate_factor(new_factor(ind, levels))
}

factor(c("a", "a", "b"))
#> [1] a a b
#> Levels: a b
```

- Algunos objetos complejos se especifican de manera más natural mediante múltiples componentes simples. Por ejemplo, creo que es natural construir una fecha y hora proporcionando los componentes individuales (año, mes, día, etc.). Eso me lleva a este ayudante `POSIXct()` que se parece a la función existente `ISODatetime()`[3]:

```
POSIXct <- function(year = integer(),
                    month = integer(),
                    day = integer(),
                    hour = 0L,
                    minute = 0L,
                    sec = 0,
```

[3]Este ayudante no es eficiente: en segundo plano `ISODatetime()` funciona pegando los componentes en una cadena y luego usando `strptime()`. Un equivalente más eficiente está disponible en `lubridate::make_datetime()`.

```
                    tzone = "") {
  ISOdatetime(year, month, day, hour, minute, sec, tz = tzone)
}

POSIXct(2020, 1, 1, tzone = "America/New_York")
#> [1] "2020-01-01 EST"
```

Para clases más complicadas, debe sentirse libre de ir más allá de estos patrones para hacer la vida lo más fácil posible para sus usuarios.

13.3.4. Ejercicios

1. Escribe un constructor para los objetos `data.frame`. ¿Sobre qué tipo base se construye un marco de datos? ¿Qué atributos utiliza? ¿Cuáles son las restricciones impuestas a los elementos individuales? ¿Qué pasa con los nombres?

2. Mejore mi ayudante `factor()` para que tenga un mejor comportamiento cuando uno o más `valores` no se encuentran en los `niveles`. ¿Qué hace `base::factor()` en esta situación?

3. Lee atentamente el código fuente de `factor()`. ¿Qué hace que mi constructor no hace?

4. Los factores tienen un atributo opcional de "contrastes". Lea la ayuda de `C()` y describa brevemente el propósito del atributo. ¿Qué tipo debe tener? Reescriba el constructor `new_factor()` para incluir este atributo.

5. Lea la documentación de `utils::as.roman()`. ¿Cómo escribirías un constructor para esta clase? ¿Necesita un validador? ¿Qué podría hacer un ayudante?

13.4. Genéricos y métodos

El trabajo de un genérico S3 es realizar el envío de métodos, es decir, encontrar la implementación específica para una clase. El envío de métodos se realiza mediante `UseMethod()`, al que todos los genéricos llaman[4]. `UseMethod()` toma dos argumentos: el nombre de la función genérica (obligatorio) y el argumento a usar para el envío del método (opcional). Si omite el segundo argumento, se enviará en función del primer argumento, que casi siempre es lo que se desea.

La mayoría de los genéricos son muy simples y consisten solo en una llamada a `UseMethod()`. Tome `mean()` por ejemplo:

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x559e98a95f58>
#> <environment: namespace:base>
```

Crear su propio genérico es igualmente simple:

```
my_new_generic <- function(x) {
  UseMethod("my_new_generic")
}
```

(Si se pregunta por qué tenemos que repetir `my_new_generic` dos veces, piense en la Section 6.2.3.)

No pasa ninguno de los argumentos del genérico a `UseMethod()`; utiliza magia profunda para pasar al método automáticamente. El proceso preciso es complicado y con frecuencia sorprendente, por lo que debe evitar realizar cualquier cálculo de forma genérica. Para conocer todos los detalles, lea detenidamente la sección Detalles técnicos en `?UseMethod`.

[4]La excepción son los genéricos internos, que se implementan en C y son el tema de la Section 13.7.2.

13.4.1. Método de envío

¿Cómo funciona `UseMethod()`? Básicamente, crea un vector de nombres de métodos, `paste0("generic", ".", c(class(x), "default"))`, y luego busca cada método potencial a su vez. Podemos ver esto en acción con `sloop::s3_dispatch()`. Le das una llamada a un genérico S3 y enumera todos los métodos posibles. Por ejemplo, ¿qué método se llama cuando imprime un objeto `Date`?

```
x <- Sys.Date()
s3_dispatch(print(x))
#> => print.Date
#>  * print.default
```

La salida aquí es simple:

- `=>` indica el método que se llama, aquí `print.Date()`
- `*` indica un método que está definido, pero no llamado, aquí `print.default()`.

La clase "predeterminada" es una **pseudoclase** especial. Esta no es una clase real, pero se incluye para que sea posible definir un respaldo estándar que se encuentra siempre que un método específico de clase no está disponible.

La esencia del envío de métodos es bastante simple, pero a medida que avanza el capítulo, verá que se vuelve progresivamente más complicado para abarcar la herencia, los tipos base, los genéricos internos y los genéricos de grupo. El siguiente código muestra un par de casos más complicados a los que volveremos en las secciones, Section 13.6 y Section 13.7.

```
x <- matrix(1:10, nrow = 2)
s3_dispatch(mean(x))
#>    mean.matrix
```

```
#>    mean.integer
#>    mean.numeric
#> => mean.default

s3_dispatch(sum(Sys.time()))
#>    sum.POSIXct
#>    sum.POSIXt
#>    sum.default
#> => Summary.POSIXct
#>    Summary.POSIXt
#>    Summary.default
#> -> sum (internal)
```

13.4.2. Encontrar métodos

`sloop::s3_dispatch()` te permite encontrar el método específico usado para una sola llamada. ¿Qué sucede si desea encontrar todos los métodos definidos para un genérico o asociados con una clase? Ese es el trabajo de `sloop::s3_methods_generic()` y `sloop::s3_methods_class()`:

```
s3_methods_generic("mean")
#> # A tibble: 7 × 4
#>   generic class      visible source
#>   <chr>   <chr>      <lgl>   <chr>
#> 1 mean    Date       TRUE    base
#> 2 mean    default    TRUE    base
#> 3 mean    difftime   TRUE    base
#> 4 mean    POSIXct    TRUE    base
#> 5 mean    POSIXlt    TRUE    base
#> 6 mean    quosure    FALSE   registered S3method
#> 7 mean    vctrs_vctr FALSE   registered S3method
```

```
s3_methods_class("ordered")
#> # A tibble: 4 × 4
#>   generic       class   visible source
#>   <chr>         <chr>   <lgl>   <chr>
#> 1 as.data.frame ordered TRUE    base
#> 2 Ops           ordered TRUE    base
#> 3 relevel       ordered FALSE   registered S3method
#> 4 Summary       ordered TRUE    base
```

13.4.3. Crear métodos

Hay dos arrugas a tener en cuenta cuando crea un nuevo método:

- Primero, solo debe escribir un método si posee el genérico o la clase. R le permitirá definir un método incluso si no lo hace, pero es de muy mala educación. En su lugar, trabaje con el autor del genérico o de la clase para agregar el método en su código.

- Un método debe tener los mismos argumentos que su genérico. Esto se aplica en los paquetes mediante `R CMD check`, pero es una buena práctica incluso si no está creando un paquete.

  Hay una excepción a esta regla: si el genérico tiene `...`, el método puede contener un superconjunto de argumentos. Esto permite que los métodos tomen argumentos adicionales arbitrarios. La desventaja de usar `...`, sin embargo, es que cualquier argumento mal escrito se tragará silenciosamente [5], como se menciona en la Section 6.6.

[5] Consulte https://github.com/hadley/ellipsis para ver una forma experimental de advertir cuando los métodos no usan todos los argumentos en `...`, lo que proporciona una posible resolución de este problema.

13.4.4. Ejercicios

1. Lea el código fuente de `t()` y `t.test()` y confirme que `t.test()` es un método genérico de S3 y no un método de S3. ¿Qué pasa si creas un objeto con la clase `test` y llamas `t()` con él? ¿Por qué?

```
x <- structure(1:10, class = "test")
t(x)
```

2. ¿Para qué genéricos tiene métodos la clase `table`?

3. ¿Para qué genéricos tiene métodos la clase `ecdf`?

4. ¿Qué base genérica tiene el mayor número de métodos definidos?

5. Lea detenidamente la documentación de `UseMethod()` y explique por qué el siguiente código devuelve los resultados que devuelve. ¿Qué dos reglas usuales de evaluación de funciones viola `UseMethod()`?

```
g <- function(x) {
  x <- 10
  y <- 10
  UseMethod("g")
}
g.default <- function(x) c(x = x, y = y)

x <- 1
y <- 1
g(x)
#> x y
#> 1 1
```

6. ¿Cuáles son los argumentos para `[`? ¿Por qué es una pregunta difícil de responder?

13.5. Estilos de objeto

Hasta ahora me he centrado en clases de estilo vectorial como `Date` y `factor`. Estos tienen la propiedad clave de que `length(x)` representa el número de observaciones en el vector. Hay tres variantes que no tienen esta propiedad:

- Los objetos de estilo de registro utilizan una lista de vectores de igual longitud para representar componentes individuales del objeto. El mejor ejemplo de esto es `POSIXlt`, que debajo del capó es una lista de 11 componentes de fecha y hora como año, mes y día. Las clases de estilo de registro anulan `longitud()` y los métodos de creación de subconjuntos para ocultar este detalle de implementación.

```
x <- as.POSIXlt(ISOdatetime(2020, 1, 1, 0, 0, 1:3))
x
#> [1] "2020-01-01 00:00:01 UTC" "2020-01-01 00:00:02 UTC"
#> [3] "2020-01-01 00:00:03 UTC"

length(x)
#> [1] 3
length(unclass(x))
#> [1] 11

x[[1]] # the first date time
#> [1] "2020-01-01 00:00:01 UTC"
unclass(x)[[1]] # the first component, the number of seconds
#> [1] 1 2 3
```

- Los marcos de datos son similares a los objetos de estilo de registro en que ambos usan listas de vectores de igual longitud. Sin embargo, los marcos de datos son conceptualmente bidimensionales y los componentes individuales se exponen fácilmente al usuario. El número de observaciones es el número de filas, no la longitud:

```
x <- data.frame(x = 1:100, y = 1:100)
length(x)
#> [1] 2
nrow(x)
#> [1] 100
```

- Los objetos escalares normalmente usan una lista para representar una sola cosa. Por ejemplo, un objeto `lm` es una lista de longitud 12 pero representa un modelo.

  ```
  mod <- lm(mpg ~ wt, data = mtcars)
  length(mod)
  #> [1] 12
  ```

  Los objetos escalares también se pueden construir sobre funciones, llamadas y entornos[6]. En general, esto es menos útil, pero puede ver aplicaciones en `stats::ecdf()`, R6 (Chapter 14) y `rlang::quo()` (Chapter 19) .

Desafortunadamente, describir el uso apropiado de cada uno de estos estilos de objeto está más allá del alcance de este libro. Sin embargo, puede obtener más información en la documentación del paquete vctrs (https://vctrs.r-lib.org); el paquete también proporciona constructores y ayudantes que facilitan la implementación de los diferentes estilos.

13.5.1. Ejercicios

1. Categorice los objetos devueltos por `lm()`, `factor()`, `table()`, `as.Date()`, `as.POSIXct()` `ecdf()`, `ordered()`, `I()` en los estilos descritos anteriormente.

[6]También puede construir un objeto encima de una lista de pares, pero todavía tengo que encontrar una buena razón para hacerlo.

2. ¿Cómo sería una función constructora para objetos `lm`, `new_lm()`? Use `?lm` y experimente para descubrir los campos obligatorios y sus tipos.

13.6. Herencia

Las clases de S3 pueden compartir el comportamiento a través de un mecanismo llamado **herencia**. La herencia está impulsada por tres ideas:

- La clase puede ser un carácter *vector*. Por ejemplo, las clases `ordered` y `POSIXct` tienen dos componentes en su clase:

```
class(ordered("x"))
#> [1] "ordered" "factor"
class(Sys.time())
#> [1] "POSIXct" "POSIXt"
```

- Si no se encuentra un método para la clase en el primer elemento del vector, R busca un método para la segunda clase (y así sucesivamente):

```
s3_dispatch(print(ordered("x")))
#>    print.ordered
#> => print.factor
#>  * print.default
s3_dispatch(print(Sys.time()))
#> => print.POSIXct
#>    print.POSIXt
#>  * print.default
```

- Un método puede delegar trabajo llamando a `NextMethod()`. Volveremos a eso muy pronto; por ahora, tenga en cuenta que `s3_dispatch()` informa delegación con `->`.

```
s3_dispatch(ordered("x")[1])
#>    [.ordered
#> => [.factor
#>    [.default
#> -> [ (internal)
s3_dispatch(Sys.time()[1])
#> => [.POSIXct
#>    [.POSIXt
#>    [.default
#> -> [ (internal)
```

Antes de continuar, necesitamos un poco de vocabulario para describir la relación entre las clases que aparecen juntas en un vector de clase. Diremos que `ordered` es una **subclase** de `factor` porque siempre aparece antes que él en el vector de clase y, a la inversa, diremos que `factor` es una **superclase** de `ordered`.

S3 no impone restricciones en la relación entre subclases y superclases, pero su vida será más fácil si impone algunas. Le recomiendo que se adhiera a dos principios simples al crear una subclase:

- El tipo base de la subclase debe ser el mismo que el de la superclase.
- Los atributos de la subclase deben ser un superconjunto de los atributos de la superclase.

`POSIXt` no se adhiere a estos principios porque `POSIXct` tiene tipo doble y `POSIXlt` tiene tipo lista. Esto significa que `POSIXt` no es una superclase, e ilustra que es bastante posible usar el sistema de herencia S3 para implementar otros estilos de código compartido (aquí `POSIXt` juega un papel más como una interfaz), pero necesitará descubra convenciones seguras usted mismo.

13.6.1. `NextMethod()`

`NextMethod()` es la parte más difícil de entender de la herencia, por lo que comenzaremos con un ejemplo concreto para el caso de uso más común: `[`. Comenzaremos creando una clase de juguete simple: una clase `secreta` que oculta su salida cuando se imprime:

```
new_secret <- function(x = double()) {
  stopifnot(is.double(x))
  structure(x, class = "secret")
}

print.secret <- function(x, ...) {
  print(strrep("x", nchar(x)))
  invisible(x)
}

x <- new_secret(c(15, 1, 456))
x
#> [1] "xx"  "x"   "xxx"
```

Esto funciona, pero el método predeterminado `[` no conserva la clase:

```
s3_dispatch(x[1])
#>    [.secret
#>    [.default
#> => [ (internal)
x[1]
#> [1] 15
```

Para arreglar esto, necesitamos proporcionar un método `[.secret`. ¿Cómo podríamos implementar este método? El enfoque ingenuo no funcionará porque nos quedaremos atrapados en un bucle infinito:

```
`[.secret` <- function(x, i) {
  new_secret(x[i])
}
```

En su lugar, necesitamos alguna forma de llamar al código `[` subyacente, es decir, la implementación que sería llamada si no tuviéramos un método `[.secret`. Un enfoque sería `unclass()` el objeto:

```
`[.secret` <- function(x, i) {
  x <- unclass(x)
  new_secret(x[i])
}
x[1]
#> [1] "xx"
```

Esto funciona, pero es ineficiente porque crea una copia de `x`. Un mejor enfoque es usar `NextMethod()`, que resuelve de manera concisa el problema de delegar al método que se habría llamado si `[.secret` no existiera:

```
`[.secret` <- function(x, i) {
  new_secret(NextMethod())
}
x[1]
#> [1] "xx"
```

Podemos ver lo que está pasando con `sloop::s3_dispatch()`:

```
s3_dispatch(x[1])
#> => [.secret
#>    [.default
#> -> [ (internal)
```

El `=>` indica que se llama a `[.secret`, pero que `NextMethod()` delega el trabajo al método interno subyacente `[`, como se muestra en `->`.

Al igual que con `UseMethod()`, la semántica precisa de `NextMethod()` es compleja. En particular, realiza un seguimiento de la lista de posibles métodos siguientes con una variable especial, lo que significa que la modificación del objeto que se envía no tendrá ningún impacto en el método que se llamará a continuación.

13.6.2. Permitir subclases

Cuando crea una clase, debe decidir si desea permitir subclases, ya que requiere algunos cambios en el constructor y una reflexión cuidadosa en sus métodos.

Para permitir subclases, el constructor principal debe tener argumentos `...` y `class`:

```
new_secret <- function(x, ..., class = character()) {
  stopifnot(is.double(x))

  structure(
    x,
    ...,
    class = c(class, "secret")
  )
}
```

Luego, el constructor de la subclase puede simplemente llamar al constructor de la clase principal con argumentos adicionales según sea necesario. Por ejemplo, imagina que queremos crear una clase supersecreta que también oculta la cantidad de caracteres:

```
new_supersecret <- function(x) {
  new_secret(x, class = "supersecret")
}

print.supersecret <- function(x, ...) {
  print(rep("xxxxx", length(x)))
  invisible(x)
}

x2 <- new_supersecret(c(15, 1, 456))
x2
#> [1] "xxxxx" "xxxxx" "xxxxx"
```

Para permitir la herencia, también debe pensar detenidamente en sus métodos, ya que ya no puede usar el constructor. Si lo hace, el método siempre devolverá la misma clase, independientemente de la entrada. Esto obliga a quien hace una subclase a hacer mucho trabajo extra.

Concretamente, esto significa que debemos revisar el método `[.secret`. Actualmente siempre devuelve un `secret()`, incluso cuando se le da un supersecreto:

```
`[.secret` <- function(x, ...) {
  new_secret(NextMethod())
}

x2[1:3]
#> [1] "xx"  "x"   "xxx"
```

Queremos asegurarnos de que `[.secret` devuelva la misma clase que `x` incluso si es una subclase. Por lo que puedo decir, no hay forma de resolver este problema usando solo la base R. En su lugar, deberá utilizar el paquete vctrs, que proporciona una solución en forma de `vctrs::vec_restore()`

genérico. Este genérico toma dos entradas: un objeto que ha perdido información de subclase y un objeto de plantilla para usar para la restauración.

Por lo general, los métodos `vec_restore()` son bastante simples: simplemente llama al constructor con los argumentos apropiados:

```
vec_restore.secret <- function(x, to, ...) new_secret(x)
vec_restore.supersecret <- function(x, to, ...) new_supersecret(x)
```

(Si su clase tiene atributos, deberá pasarlos de `to` al constructor).

Ahora podemos usar `vec_restore()` en el método `[.secret`:

```
`[.secret` <- function(x, ...) {
  vctrs::vec_restore(NextMethod(), x)
}
x2[1:3]
#> [1] "xxxxx" "xxxxx" "xxxxx"
```

(Solo entendí completamente este problema recientemente, por lo que al momento de escribir no se usa en el tidyverse. Con suerte, para cuando estés leyendo esto, se habrá implementado, lo que hará que sea mucho más fácil (por ejemplo) subclasificar tibbles.)

Si construye su clase usando las herramientas provistas por el paquete vctrs, `[` obtendrá este comportamiento automáticamente. Solo necesitará proporcionar su propio método `[` si usa atributos que dependen de los datos o desea un comportamiento de subconjunto no estándar. Ver `?vctrs::new_vctr` para más detalles.

13.6.3. Ejercicios

1. ¿Cómo admite subclases `[.Date`? ¿Cómo no admite subclases?

2. R tiene dos clases para representar datos de fecha y hora, `POSIXct` y `POSIXlt`, que heredan ambas de `POSIXt`. ¿Qué genéricos tienen comportamientos diferentes para las dos clases? ¿Qué genéricos comparten el mismo comportamiento?

3. ¿Qué espera que devuelva este código? ¿Qué devuelve realmente? ¿Por qué?

```
generic2 <- function(x) UseMethod("generic2")
generic2.a1 <- function(x) "a1"
generic2.a2 <- function(x) "a2"
generic2.b <- function(x) {
  class(x) <- "a1"
  NextMethod()
}

generic2(structure(list(), class = c("b", "a2")))
```

13.7. Detalles de envío

Este capítulo concluye con algunos detalles adicionales sobre el envío de métodos. Es seguro omitir estos detalles si es nuevo en S3.

13.7.1. S3 y tipos básicos

¿Qué sucede cuando llama a un genérico S3 con un objeto base, es decir, un objeto sin clase? Podrías pensar que enviaría lo que `class()` devuelve:

```
class(matrix(1:5))
#> [1] "matrix" "array"
```

Pero, lamentablemente, el envío se produce en la **clase implícita**, que tiene tres componentes:

- La cadena "array" o "matrix" si el objeto tiene dimensiones
- El resultado de `typeof()` con algunos ajustes menores
- La cadena "numeric" si el objeto es "integer" o "double"

No hay una función base que calcule la clase implícita, pero puede usar `sloop::s3_class()`

```
s3_class(matrix(1:5))
#> [1] "matrix"  "integer" "numeric"
```

Esto es usado por `s3_dispatch()`:

```
s3_dispatch(print(matrix(1:5)))
#>    print.matrix
#>    print.integer
#>    print.numeric
#> => print.default
```

Esto significa que la clade, `class()`, de un objeto no determina de forma única su envío:

```
x1 <- 1:5
class(x1)
#> [1] "integer"
s3_dispatch(mean(x1))
#>    mean.integer
```

```
#>    mean.numeric
#> => mean.default

x2 <- structure(x1, class = "integer")
class(x2)
#> [1] "integer"
s3_dispatch(mean(x2))
#>    mean.integer
#> => mean.default
```

13.7.2. Genéricos internos

Algunas funciones básicas, como `[`, `sum()` y `cbind()`, se denominan **genéricos internos** porque no llaman a `UseMethod()` sino que llaman a las funciones de C `DispatchGroup( )` o `DispatchOrEval()`. `s3_dispatch()` muestra genéricos internos al incluir el nombre del genérico seguido de `(internal)`:

```
s3_dispatch(Sys.time()[1])
#> => [.POSIXct
#>    [.POSIXt
#>    [.default
#> -> [ (internal)
```

Por motivos de rendimiento, los genéricos internos no envían a los métodos a menos que se haya establecido el atributo de clase, lo que significa que los genéricos internos no utilizan la clase implícita. Nuevamente, si alguna vez se siente confundido acerca del envío de métodos, puede confiar en `s3_dispatch()`.

13.7.3. Genéricos del grupo

Los genéricos de grupo son la parte más complicada del envío de métodos de S3 porque involucran tanto `NextMethod()` como genéricos internos. Al igual que los genéricos internos, solo existen en la base R y no puede definir su propio grupo genérico.

Hay cuatro genéricos de grupo:

- **Matemáticas**: `abs()`, `sign()`, `sqrt()`, `floor()`, `cos()`, `sin()`, `log()`, y más (ver `?Math` para la lista completa).
- **Operaciones**: `+`, `-`, `*`, `/`, `^`, `%%`, `%/%`, `&`, `|`, `!`, `==`, `!=`, `<`, `<=`, `>=`, y `>`.
- **Resumen**: `all()`, `any()`, `sum()`, `prod()`, `min()`, `max()`, y `range()`.
- **Complejo**: `Arg()`, `Conj()`, `Im()`, `Mod()`, `Re()`.

La definición de un solo grupo genérico para su clase anula el comportamiento predeterminado para todos los miembros del grupo. Los métodos para genéricos grupales se buscan solo si los métodos para el genérico específico no existen:

```
s3_dispatch(sum(Sys.time()))
#>    sum.POSIXct
#>    sum.POSIXt
#>    sum.default
#> => Summary.POSIXct
#>    Summary.POSIXt
#>    Summary.default
#> -> sum (internal)
```

La mayoría de los genéricos de grupo implican una llamada a `NextMethod()`. Por ejemplo, tome los objetos `difftime()`. Si observa el envío del método para `abs()`, verá que hay un grupo genérico `Math` definido.

```
y <- as.difftime(10, units = "mins")
s3_dispatch(abs(y))
#>    abs.difftime
#>    abs.default
#> => Math.difftime
#>    Math.default
#> -> abs (internal)
```

`Math.difftime` básicamente se ve así:

```
Math.difftime <- function(x, ...) {
  new_difftime(NextMethod(), units = attr(x, "units"))
}
```

Despacha al siguiente método, aquí el valor predeterminado interno, para realizar el cálculo real y luego restaurar la clase y los atributos. (Para admitir mejor las subclases de `difftime`, sería necesario llamar a `vec_restore()`, como se describe en la Section 13.6.2.)

Dentro de una función genérica de grupo, una variable especial `.Generic` proporciona la función genérica real llamada. Esto puede ser útil cuando se producen mensajes de error y, a veces, puede ser útil si necesita recuperar manualmente el genérico con diferentes argumentos.

13.7.4. Despacho doble

Los genéricos del grupo Ops, que incluye la aritmética de dos argumentos y los operadores booleanos como `-` y `&`, implementan un tipo especial de envío de métodos. Despachan en el tipo de *ambos* argumentos, que se llama **despacho doble**. Esto es necesario para preservar la propiedad conmutativa de muchos operadores, es decir, `a + b` debería ser igual a `b + a`. Tome el siguiente ejemplo simple:

```
date <- as.Date("2017-01-01")
integer <- 1L

date + integer
#> [1] "2017-01-02"
integer + date
#> [1] "2017-01-02"
```

Si + se enviara solo en el primer argumento, devolvería valores diferentes para los dos casos. Para superar este problema, los genéricos del grupo Ops utilizan una estrategia ligeramente diferente a la habitual. En lugar de hacer un envío de un solo método, hacen dos, uno para cada entrada. Hay tres posibles resultados de esta búsqueda:

- Los métodos son los mismos, por lo que no importa qué método se utilice.
- Los métodos son diferentes y R recurre al método interno con una advertencia.
- Un método es interno, en cuyo caso R llama al otro método.

Este enfoque es propenso a errores, por lo que si desea implementar un despacho doble robusto para operadores algebraicos, le recomiendo usar el paquete vctrs. Ver `?vctrs::vec_arith` para más detalles.

13.7.5. Ejercicios

1. Explique las diferencias en el envío a continuación:

```
length.integer <- function(x) 10

x1 <- 1:5
class(x1)
```

```
#> [1] "integer"
s3_dispatch(length(x1))
#>  * length.integer
#>    length.numeric
#>    length.default
#> => length (internal)

x2 <- structure(x1, class = "integer")
class(x2)
#> [1] "integer"
s3_dispatch(length(x2))
#> => length.integer
#>    length.default
#>  * length (internal)
```

2. ¿Qué clases tienen un método para el grupo `Math` genérico en base R? Lee el código fuente. ¿Cómo funcionan los métodos?

3. `Math.difftime()` es más complicado de lo que describí. ¿Por qué?

14. R6

14.1. Introducción

Este capítulo describe el sistema R6 OOP. R6 tiene dos propiedades especiales:

- Utiliza el paradigma OOP encapsulado, lo que significa que los métodos pertenecen a los objetos, no a los genéricos, y los llamas como `object$method()`.

- Los objetos R6 son **mutables**, lo que significa que se modifican en su lugar y, por lo tanto, tienen semántica de referencia.

Si aprendió programación orientada a objetos en otro lenguaje de programación, es probable que R6 se sienta muy natural y se incline a preferirlo a S3. Resista la tentación de seguir el camino de menor resistencia: en la mayoría de los casos, R6 lo llevará a un código R no idiomático. Volveremos a este tema en la Section 16.3.

R6 es muy similar a un sistema OOP base llamado **clases de referencia**, o RC para abreviar. Describo por qué enseño R6 y no RC en la Section 14.5.

Estructura

- La Section 14.2 introduce `R6::R6Class()`, la única función que necesita saber para crear clases R6. Aprenderá sobre el método constructor, `$new()`, que le permite crear objetos R6, así como otros métodos importantes como `$initialize()` y `$print()`.
- La Section 14.3 analiza los mecanismos de acceso de R6: campos privados y activos. Juntos, estos le permiten ocultar datos del usuario o exponer datos privados para leer pero no escribir.
- La Section 14.4 explora las consecuencias de la semántica de referencia de R6. Aprenderá sobre el uso de finalizadores para limpiar automáticamente cualquier operación realizada en el inicializador y un problema común si usa un objeto R6 como un campo en otro objeto R6.
- La Section 14.5 describe por qué cubro R6, en lugar del sistema RC base.

Requisitos previos

Debido a que R6 no está integrado en la base R, deberá instalar y cargar el paquete R6 para usarlo:

```
# install.packages("R6")
library(R6)
```

Los objetos R6 tienen semántica de referencia, lo que significa que se modifican en el lugar, no se copian al modificar. Si no está familiarizado con estos términos, repase su vocabulario leyendo la Section 2.5.

14.2. Clases y métodos

R6 solo necesita una única llamada de función para crear tanto la clase como sus métodos: `R6::R6Class()`. ¡Esta es la única función del paquete que usará![1]

El siguiente ejemplo muestra los dos argumentos más importantes para `R6Class()`:

- El primer argumento es el `classname`. No es estrictamente necesario, pero mejora los mensajes de error y permite usar objetos R6 con genéricos S3. Por convención, las clases R6 tienen nombres `UpperCamelCase`.

- El segundo argumento, `public`, proporciona una lista de métodos (funciones) y campos (cualquier otra cosa) que conforman la interfaz pública del objeto. Por convención, los métodos y campos usan `snake_case`. Los métodos pueden acceder a los métodos y campos del objeto actual a través de `self$`.[2]

```
Accumulator <- R6Class("Accumulator", list(
  sum = 0,
  add = function(x = 1) {
    self$sum <- self$sum + x
    invisible(self)
  })
)
```

[1] Eso significa que si está creando R6 en un paquete, solo necesita asegurarse de que esté listado en el campo `Imports` de `DESCRIPCIÓN`. No hay necesidad de importar el paquete a `NAMESPACE`.

[2] A diferencia de Python, R6 proporciona automáticamente la variable `self` y no forma parte de la firma del método.

Siempre debe asignar el resultado de `R6Class()` a una variable con el mismo nombre que la clase, porque `R6Class()` devuelve un objeto R6 que define la clase:

```
Accumulator
#> <Accumulator> object generator
#>   Public:
#>     sum: 0
#>     add: function (x = 1)
#>     clone: function (deep = FALSE)
#>   Parent env: <environment: R_GlobalEnv>
#>   Locked objects: TRUE
#>   Locked class: FALSE
#>   Portable: TRUE
```

Construyes un nuevo objeto a partir de la clase llamando al método `new()`. En R6, los métodos pertenecen a los objetos, por lo que usa `$` para acceder a `new()`:

```
x <- Accumulator$new()
```

A continuación, puede llamar a los métodos y acceder a los campos con `$`:

```
x$add(4)
x$sum
#> [1] 4
```

En esta clase, los campos y métodos son públicos, lo que significa que puede obtener o establecer el valor de cualquier campo. Más adelante, veremos cómo usar campos y métodos privados para evitar el acceso casual a las partes internas de su clase.

Para que quede claro cuando hablamos de campos y métodos en lugar de variables y funciones, pondré el prefijo `$` en sus nombres. Por ejemplo, la clase `Accumulate` tiene el campo `$sum` y el método `$add()`.

14.2.1. Encadenamiento de métodos

`$add()` se llama principalmente por su efecto secundario de actualizar `$sum`.

```
Accumulator <- R6Class("Accumulator", list(
  sum = 0,
  add = function(x = 1) {
    self$sum <- self$sum + x
    invisible(self)
  })
)
```

Los métodos R6 de efectos secundarios siempre deben devolver `self` de forma invisible. Esto devuelve el objeto "actual" y hace posible encadenar varias llamadas a métodos:

```
x$add(10)$add(10)$sum
#> [1] 24
```

Para facilitar la lectura, puede poner una llamada de método en cada línea:

```
x$
  add(10)$
  add(10)$
  sum
#> [1] 44
```

Esta técnica se llama **encadenamiento de métodos** y se usa comúnmente en lenguajes como Python y JavaScript. El encadenamiento de métodos está profundamente relacionado con la tubería, y discutiremos los pros y los contras de cada enfoque en la Section 16.3.3.

14.2.2. Métodos importantes

Hay dos métodos importantes que deben definirse para la mayoría de las clases: `$initialize()` y `$print()`. No son obligatorios, pero proporcionarlos hará que su clase sea más fácil de usar.

`$initialize()` anula el comportamiento predeterminado de `$new()`. Por ejemplo, el siguiente código define una clase de Persona con los campos `$name` y `$age`. Para asegurar que `$name` sea siempre una sola cadena, y `$age` sea siempre un solo número, puse controles en `$initialize()`.

```
Person <- R6Class("Person", list(
  name = NULL,
  age = NA,
  initialize = function(name, age = NA) {
    stopifnot(is.character(name), length(name) == 1)
    stopifnot(is.numeric(age), length(age) == 1)

    self$name <- name
    self$age <- age
  }
))

hadley <- Person$new("Hadley", age = "thirty-eight")
#> Error in initialize(...): is.numeric(age) is not TRUE

hadley <- Person$new("Hadley", age = 38)
```

Si tiene requisitos de validación más costosos, impleméntelos en un `$validate()` separado y solo llame cuando sea necesario.

Definir `$print()` le permite anular el comportamiento de impresión predeterminado. Como con cualquier método R6 llamado por sus efectos secundarios, `$print()` debería devolver `invisible(self)`.

```
Person <- R6Class("Person", list(
  name = NULL,
  age = NA,
  initialize = function(name, age = NA) {
    self$name <- name
    self$age <- age
  },
  print = function(...) {
    cat("Person: \n")
    cat("  Name: ", self$name, "\n", sep = "")
    cat("  Age:  ", self$age, "\n", sep = "")
    invisible(self)
  }
))

hadley2 <- Person$new("Hadley")
hadley2
#> Person:
#>   Name: Hadley
#>   Age:  NA
```

Este código ilustra un aspecto importante de R6. Debido a que los métodos están vinculados a objetos individuales, el objeto `hadley` creado previamente no obtiene este nuevo método:

```
hadley
#> <Person>
#>   Public:
#>     age: 38
#>     clone: function (deep = FALSE)
#>     initialize: function (name, age = NA)
#>     name: Hadley

hadley$print
#> NULL
```

Desde la perspectiva de R6, no hay relación entre `hadley` y `hadley2`; coincidentemente comparten el mismo nombre de clase. Esto no causa problemas cuando se usan objetos R6 ya desarrollados, pero puede hacer que la experimentación interactiva sea confusa. Si está cambiando el código y no puede averiguar por qué los resultados de las llamadas a métodos no son diferentes, asegúrese de haber reconstruido los objetos R6 con la nueva clase.

14.2.3. Agregar métodos después de la creación

En lugar de crear continuamente nuevas clases, también es posible modificar los campos y métodos de una clase existente. Esto es útil al explorar de forma interactiva o cuando tiene una clase con muchas funciones que le gustaría dividir en partes. Agrega nuevos elementos a una clase existente con `$set()`, proporcionando la visibilidad (más información en la Section 14.3), el nombre y el componente.

```
Accumulator <- R6Class("Accumulator")
Accumulator$set("public", "sum", 0)
Accumulator$set("public", "add", function(x = 1) {
  self$sum <- self$sum + x
```

```
  invisible(self)
})
```

Como se indicó anteriormente, los nuevos métodos y campos solo están disponibles para nuevos objetos; no se agregan retrospectivamente a los objetos existentes.

14.2.4. Herencia

Para heredar el comportamiento de una clase existente, proporcione el objeto de la clase al argumento `inherit`:

```
AccumulatorChatty <- R6Class("AccumulatorChatty",
  inherit = Accumulator,
  public = list(
    add = function(x = 1) {
      cat("Adding ", x, "\n", sep = "")
      super$add(x = x)
    }
  )
)

x2 <- AccumulatorChatty$new()
x2$add(10)$add(1)$sum
#> Adding 10
#> Adding 1
#> [1] 11
```

`$add()` anula la implementación de la superclase, pero aún podemos delegar a la implementación de la superclase usando `super$`. (Esto es análogo a `NextMethod()` en S3, como se explica en la Section 13.6.) Cualquier método que no se invalide utilizará la implementación en la clase principal.

14.2.5. Introspección

Cada objeto R6 tiene una clase S3 que refleja su jerarquía de clases R6. Esto significa que la forma más fácil de determinar la clase (y todas las clases de las que hereda) es usar `class()`:

```
class(hadley2)
#> [1] "Person" "R6"
```

La jerarquía S3 incluye la clase base "R6". Esto proporciona un comportamiento común, incluido un método `print.R6()` que llama a `$print()`, como se describe arriba.

Puede enumerar todos los métodos y campos con `names()`:

```
names(hadley2)
#> [1] ".__enclos_env__" "age"             "name"
#> [4] "clone"           "print"           "initialize"
```

Definimos `$name`, `$age`, `$print` e `$initialize`. Como sugiere el nombre, `.__enclos_env__` es un detalle de implementación interna que no debe tocar; volveremos a `$clone()` en la Section 14.4.

14.2.6. Ejercicios

1. Cree una cuenta bancaria clase R6 que almacene un saldo y le permita depositar y retirar dinero. Cree una subclase que arroje un error si intenta entrar en sobregiro. Cree otra subclase que le permita entrar en sobregiro, pero le cobre una tarifa.

2. Cree una clase R6 que represente un mazo de cartas barajado. Deberías poder sacar cartas del mazo con `$draw(n)`, devolver todas las cartas al mazo y volver a barajar con `$reshuffle()`. Use el siguiente código para hacer un vector de tarjetas.

```
suit <- c(" ", " ", " ", " ")
value <- c("A", 2:10, "J", "Q", "K")
cards <- paste0(rep(value, 4), suit)
```

3. ¿Por qué no puedes modelar una cuenta bancaria o una baraja de cartas con una clase S3?

4. Cree una clase R6 que le permita obtener y establecer la zona horaria actual. Puede acceder a la zona horaria actual con `Sys.timezone()` y configurarla con `Sys.setenv(TZ = "newtimezone")`. Al configurar la zona horaria, asegúrese de que la nueva zona horaria esté en la lista proporcionada por `OlsonNames()`.

5. Cree una clase R6 que administre el directorio de trabajo actual. Debe tener los métodos `$get()` y `$set()`.

6. ¿Por qué no puede modelar la zona horaria o el directorio de trabajo actual con una clase S3?

7. ¿Sobre qué tipo base se construyen los objetos R6? ¿Qué atributos tienen?

14.3. Control de acceso

`R6Class()` tiene otros dos argumentos que funcionan de manera similar a `public`:

- `private` te permite crear campos y métodos que solo están disponibles dentro de la clase, no fuera de ella.
- `activo` le permite usar funciones de acceso para definir campos dinámicos o activos.

Estos se describen en las siguientes secciones.

14.3.1. Privacidad

Con R6 puedes definir campos y métodos **privados**, elementos a los que solo se puede acceder desde dentro de la clase, no desde fuera[3]. Hay dos cosas que debe saber para aprovechar los elementos privados:

- El argumento 'privado' de 'R6Class' funciona de la misma manera que el argumento 'publico': le da una lista con nombre de métodos (funciones) y campos (todo lo demás).
- Los campos y métodos definidos en `private` están disponibles dentro de los métodos que usan `private$` en lugar de `self$`. No puede acceder a campos o métodos privados fuera de la clase.

Para concretar esto, podríamos hacer que los campos `$age` y `$name` de la clase Persona sean privados. Con esta definición de `Person` solo podemos establecer `$age` y `$name` durante la creación del objeto, y no podemos acceder a sus valores desde fuera de la clase.

```
Person <- R6Class("Person",
  public = list(
    initialize = function(name, age = NA) {
      private$name <- name
      private$age <- age
    },
    print = function(...) {
      cat("Person: \n")
      cat("  Name: ", private$name, "\n", sep = "")
      cat("  Age:  ", private$age, "\n", sep = "")
    }
  ),
```

[3]Debido a que R es un lenguaje tan flexible, técnicamente aún es posible acceder a valores privados, pero tendrá que esforzarse mucho más, profundizando en los detalles de la implementación de R6.

```
  private = list(
    age = NA,
    name = NULL
  )
)

hadley3 <- Person$new("Hadley")
hadley3
#> Person:
#>   Name: Hadley
#>   Age:  NA
hadley3$name
#> NULL
```

La distinción entre campos públicos y privados es importante cuando crea redes complejas de clases y desea dejar lo más claro posible qué está bien que otros accedan. Cualquier cosa que sea privada puede refactorizarse más fácilmente porque sabe que otros no confían en ella. Los métodos privados tienden a ser menos importantes en R en comparación con otros lenguajes de programación porque las jerarquías de objetos en R tienden a ser más simples.

14.3.2. Campos activos

Los campos activos le permiten definir componentes que parecen campos desde el exterior, pero se definen con funciones, como métodos. Los campos activos se implementan mediante **enlaces activos** (Section 7.2.6). Cada enlace activo es una función que toma un único argumento: `value`. Si el argumento es `missing()`, se está recuperando el valor; de lo contrario, se está modificando.

Por ejemplo, podría crear un campo activo `random` que devuelva un valor diferente cada vez que acceda a él:

```
Rando <- R6::R6Class("Rando", active = list(
  random = function(value) {
    if (missing(value)) {
      runif(1)
    } else {
      stop("Can't set `$random`", call. = FALSE)
    }
  }
))
x <- Rando$new()
x$random
#> [1] 0.0808
x$random
#> [1] 0.834
x$random
#> [1] 0.601
```

Los campos activos son especialmente útiles junto con los campos privados, ya que permiten implementar componentes que parecen campos desde el exterior pero proporcionan comprobaciones adicionales. Por ejemplo, podemos usarlos para crear un campo `age` de solo lectura y para asegurarnos de que `name` sea un vector de caracteres de longitud 1.

```
Person <- R6Class("Person",
  private = list(
    .age = NA,
    .name = NULL
  ),
  active = list(
    age = function(value) {
      if (missing(value)) {
        private$.age
      } else {
```

```
        stop("`$age` is read only", call. = FALSE)
      }
    },
    name = function(value) {
      if (missing(value)) {
        private$.name
      } else {
        stopifnot(is.character(value), length(value) == 1)
        private$.name <- value
        self
      }
    }
  ),
  public = list(
    initialize = function(name, age = NA) {
      private$.name <- name
      private$.age <- age
    }
  )
)

hadley4 <- Person$new("Hadley", age = 38)
hadley4$name
#> [1] "Hadley"
hadley4$name <- 10
#> Error in (function (value) : is.character(value) is not TRUE
hadley4$age <- 20
#> Error: `$age` is read only
```

14.3.3. Ejercicios

1. Cree una clase de cuenta bancaria que le impida establecer directamente el saldo de la cuenta, pero aún puede retirar y depositar. Lanza un error si intentas entrar en sobregiro.
2. Cree una clase con un campo `$password` de solo escritura. Debería tener el método `$check_password(password)` que devuelva `TRUE` o `FALSE`, pero no debería haber forma de ver la contraseña completa.
3. Extienda la clase `Rando` con otro enlace activo que le permita acceder al valor aleatorio anterior. Asegúrese de que el enlace activo sea la única forma de acceder al valor.
4. ¿Pueden las subclases acceder a campos/métodos privados desde su padre? Haz un experimento para averiguarlo.

14.4. Semántica de referencia

Una de las grandes diferencias entre R6 y la mayoría de los demás objetos es que tienen semántica de referencia. La consecuencia principal de la semántica de referencia es que los objetos no se copian cuando se modifican:

```
y1 <- Accumulator$new()
y2 <- y1

y1$add(10)
c(y1 = y1$sum, y2 = y2$sum)
#> y1 y2
#> 10 10
```

En cambio, si desea una copia, deberá explícitamente `$clone()` el objeto:

```
y1 <- Accumulator$new()
y2 <- y1$clone()

y1$add(10)
c(y1 = y1$sum, y2 = y2$sum)
#> y1 y2
#> 10  0
```

(`$clone()` no clona recursivamente objetos R6 anidados. Si quieres eso, tendrás que usar `$clone(deep = TRUE)`.)

Hay otras tres consecuencias menos obvias:

- Es más difícil razonar sobre el código que usa objetos R6 porque necesita comprender más contexto.
- Tiene sentido pensar en cuándo se elimina un objeto R6 y puede escribir `$finalize()` para complementar el `$initialize()`.
- Si uno de los campos es un objeto R6, debe crearlo dentro `$initialize()`, no `R6Class()`.

Estas consecuencias se describen con más detalle a continuación.

14.4.1. Razonamiento

En general, la semántica de referencia hace que sea más difícil razonar sobre el código. Tome este ejemplo muy simple:

```
x <- list(a = 1)
y <- list(b = 2)

z <- f(x, y)
```

Para la gran mayoría de las funciones, sabes que la línea final solo modifica `z`.

Tome un ejemplo similar que usa una clase de referencia `List` imaginaria:

```
x <- List$new(a = 1)
y <- List$new(b = 2)

z <- f(x, y)
```

La línea final es mucho más difícil de razonar: si `f()` llama a métodos de `x` o `y`, podría modificarlos así como `z`. Este es el mayor inconveniente potencial de R6 y debe tener cuidado de evitarlo escribiendo funciones que devuelvan un valor o modifiquen sus entradas R6, pero no ambos. Dicho esto, hacer ambas cosas puede conducir a un código sustancialmente más simple en algunos casos, y discutiremos esto más adelante en la Section 16.3.2).

14.4.2. Finalizador

Una propiedad útil de la semántica de referencia es que tiene sentido pensar cuándo se **finaliza** un objeto R6, es decir, cuándo se elimina. Esto no tiene sentido para la mayoría de los objetos porque la semántica de copiar al modificar significa que puede haber muchas versiones transitorias de un objeto, como se menciona en la Section 2.6. Por ejemplo, lo siguiente crea dos objetos de factor: el segundo se crea cuando se modifican los niveles, dejando que el primero sea destruido por el recolector de basura.

```
x <- factor(c("a", "b", "c"))
levels(x) <- c("c", "b", "a")
```

Dado que los objetos R6 no se copian al modificarse, solo se eliminan una vez, y tiene sentido pensar en `$finalize()` como un complemento de

`$initialize()`. Los finalizadores generalmente juegan un papel similar a `on.exit()` (como se describe en la Section 6.7.4), limpiando cualquier recurso creado por el inicializador. Por ejemplo, la siguiente clase envuelve un archivo temporal y lo elimina automáticamente cuando finaliza la clase.

```
TemporaryFile <- R6Class("TemporaryFile", list(
  path = NULL,
  initialize = function() {
    self$path <- tempfile()
  },
  finalize = function() {
    message("Cleaning up ", self$path)
    unlink(self$path)
  }
))
```

El método finalize se ejecutará cuando se elimine el objeto (o más precisamente, por la primera recolección de elementos no utilizados después de que el objeto se haya desvinculado de todos los nombres) o cuando R salga. Esto significa que el finalizador se puede llamar de manera efectiva en cualquier parte de su código R y, por lo tanto, es casi imposible razonar sobre el código del finalizador que toca las estructuras de datos compartidas. Evite estos posibles problemas utilizando únicamente el finalizador para limpiar los recursos privados asignados por el inicializador.

```
tf <- TemporaryFile$new()
rm(tf)
#> Cleaning up /tmp/Rtmpk73JdI/file155f31d8424bd
```

14.4.3. Campos de R6

Una consecuencia final de la semántica de referencia puede surgir donde no lo espera. Si usa una clase R6 como el valor predeterminado de un campo, ¡se compartirá entre todas las instancias del objeto! Toma el siguiente código: queremos crear una base de datos temporal cada vez que llamamos a `TemporaryDatabase$new()`, pero el código actual siempre usa la misma ruta.

```
TemporaryDatabase <- R6Class("TemporaryDatabase", list(
  con = NULL,
  file = TemporaryFile$new(),
  initialize = function() {
    self$con <- DBI::dbConnect(RSQLite::SQLite(), path = file$path)
  },
  finalize = function() {
    DBI::dbDisconnect(self$con)
  }
))

db_a <- TemporaryDatabase$new()
db_b <- TemporaryDatabase$new()

db_a$file$path == db_b$file$path
#> [1] TRUE
```

(Si está familiarizado con Python, esto es muy similar al problema del "argumento predeterminado mutable").

El problema surge porque `TemporaryFile$new()` se llama solo una vez cuando se define la clase `TemporaryDatabase`. Para solucionar el problema, debemos asegurarnos de que se llame cada vez que se llame a `TemporaryDatabase$new()`, es decir, debemos ponerlo en `$initialize()`:

```
TemporaryDatabase <- R6Class("TemporaryDatabase", list(
  con = NULL,
  file = NULL,
  initialize = function() {
    self$file <- TemporaryFile$new()
    self$con <- DBI::dbConnect(RSQLite::SQLite(), path = file$path)
  },
  finalize = function() {
    DBI::dbDisconnect(self$con)
  }
))

db_a <- TemporaryDatabase$new()
db_b <- TemporaryDatabase$new()

db_a$file$path == db_b$file$path
#> [1] FALSE
```

14.4.4. Ejercicios

1. Cree una clase que le permita escribir una línea en un archivo específico. Debe abrir una conexión al archivo en `$initialize()`, agregar una línea usando `cat()` en `$append_line()` y cerrar la conexión en `$finalize()`.

14.5. ¿Por qué R6?

R6 es muy similar a un sistema OO incorporado llamado **clases de referencia**, o RC para abreviar. Prefiero R6 a RC porque:

- R6 es mucho más simple. Tanto R6 como RC están construidos sobre entornos, pero mientras que R6 usa S3, RC usa S4. Esto significa que para comprender completamente RC, debe comprender cómo funciona el S4 más complicado.
- R6 tiene documentación completa en línea en https://r6.r-lib.org.
- R6 tiene un mecanismo más simple para la subclasificación de paquetes cruzados, que simplemente funciona sin que tengas que pensar en ello. Para RC, lea los detalles en la sección "Métodos externos; Superclases entre paquetes" de `?setRefClass`.
- RC mezcla variables y campos en la misma pila de entornos para que obtenga (`field`) y establezca (`field <<- value`) campos como valores regulares. R6 coloca los campos en un entorno separado para que obtenga (`self$field`) y establezca (`self$field <- value`) con un prefijo. El enfoque R6 es más detallado, pero me gusta porque es más explícito.
- R6 es mucho más rápido que RC. En general, la velocidad de envío del método no es importante fuera de los micropuntos de referencia. Sin embargo, RC es bastante lento y cambiar de RC a R6 condujo a una mejora sustancial del rendimiento en el paquete brillante. Para obtener más detalles, consulte `vignette("Rendimiento", "R6")`.
- RC está vinculado a R. Eso significa que si se corrigen errores, solo puede aprovechar las correcciones al solicitar una versión más nueva de R. Esto dificulta los paquetes (como los del tidyverse) que necesitan funcionar en muchos R versiones.
- Finalmente, debido a que las ideas que subyacen en R6 y RC son similares, solo requerirá una pequeña cantidad de esfuerzo adicional para aprender RC si es necesario.

15. S4

15.1. Introducción

S4 proporciona un enfoque formal para la programación orientada a objetos funcional. Las ideas subyacentes son similares a S3 (el tema del Chapter 13), pero la implementación es mucho más estricta y utiliza funciones especializadas para crear clases (`setClass()`), genéricos (`setGeneric()`) y métodos (`setMethod()`). Además, S4 proporciona herencia múltiple (es decir, una clase puede tener varios padres) y envío múltiple (es decir, el envío del método puede usar la clase de varios argumentos).

Un nuevo componente importante de S4 es la **ranura**, un componente con nombre del objeto al que se accede mediante el operador de subconjunto especializado `@`. El conjunto de ranuras y sus clases forma una parte importante de la definición de una clase S4.

Estructura

- La Section 15.2 brinda una descripción general rápida de los componentes principales de S4: clases, genéricos y métodos.
- La Section 15.3 se sumerge en los detalles de las clases de S4, incluidos prototipos, constructores, ayudantes y validadores.
- La Section 15.4 le muestra cómo crear nuevos genéricos S4 y cómo proporcionar métodos a esos genéricos. También aprenderá acerca de las funciones de acceso que están diseñadas para permitir que los

usuarios inspeccionen y modifiquen las ranuras de objetos de manera segura.

- La Section 15.5 se sumerge en los detalles completos del envío de métodos en S4. La idea básica es simple, pero rápidamente se vuelve más compleja una vez que se combinan la herencia múltiple y el envío múltiple.
- La Section 15.6 analiza la interacción entre S4 y S3 y le muestra cómo usarlos juntos.

Aprendiendo más

Al igual que los otros capítulos de OO, el enfoque aquí será cómo funciona S4, no cómo implementarlo de manera más efectiva. Si desea usarlo en la práctica, hay dos desafíos principales:

- No hay una referencia que responda a todas sus preguntas sobre S4.
- La documentación integrada de R a veces choca con las mejores prácticas de la comunidad.

A medida que avanza hacia un uso más avanzado, deberá reunir la información necesaria leyendo detenidamente la documentación, haciendo preguntas sobre StackOverflow y realizando experimentos. Algunas recomendaciones:

- La comunidad de bioconductores es un usuario de S4 desde hace mucho tiempo y ha producido gran parte del mejor material sobre su uso efectivo. Comience con Clases y métodos de S4 impartido por Martin Morgan y Hervé Pagès, o busque uno más nuevo versión en [Materiales del curso de bioconductores](https://bioconductor.org/help/course-materials/).

  Martin Morgan es un ex miembro de R-core y líder del proyecto de Bioconductor. Es un experto mundial en el uso práctico de S4 y

recomiendo leer todo lo que ha escrito al respecto, comenzando con las preguntas que ha respondido en stackoverflow.

- John Chambers es el autor del sistema S4 y proporciona una descripción general de su motivación y contexto histórico en *Programación orientada a objetos, programación funcional y R* (Chambers 2014). Para una exploración más completa de S4, consulte su libro *Software for Data Analysis* (Chambers 2008).

Requisitos previos

Todas las funciones relacionadas con S4 viven en el paquete de métodos. Este paquete siempre está disponible cuando ejecuta R de forma interactiva, pero es posible que no esté disponible cuando ejecuta R en modo por lotes, es decir, desde `Rscript`[1]. Por esta razón, es una buena idea llamar a `library(methods)` siempre que use S4. Esto también indica al lector que utilizará el sistema de objetos S4.

```
library(methods)
```

15.2. Lo escencial

Comenzaremos con una descripción general rápida de los componentes principales de S4. Una clase de S4 se define llamando a `setClass()` con el nombre de la clase y una definición de sus ranuras, y los nombres y clases de los datos de la clase:

[1] Esta es una peculiaridad histórica introducida porque el paquete de métodos solía tardar mucho tiempo en cargarse y `Rscript` está optimizado para una invocación rápida de la línea de comandos.

```
setClass("Person",
  slots = c(
    name = "character",
    age = "numeric"
  )
)
```

Una vez que se define la clase, puede construir nuevos objetos a partir de ella llamando a `new()` con el nombre de la clase y un valor para cada ranura:

```
john <- new("Person", name = "John Smith", age = NA_real_)
```

Dado un objeto S4, puede ver su clase con `is()` y acceder a las ranuras con `@` (equivalente a `$`) y `slot()` (equivalente a `[[`):

```
is(john)
#> [1] "Person"
john@name
#> [1] "John Smith"
slot(john, "age")
#> [1] NA
```

En general, solo debe usar `@` en sus métodos. Si está trabajando con la clase de otra persona, busque funciones de **accesorio** que le permitan establecer y obtener valores de ranura de forma segura. Como desarrollador de una clase, también debe proporcionar sus propias funciones de acceso. Los accesores suelen ser genéricos de S4 que permiten que varias clases compartan la misma interfaz externa.

Aquí crearemos un setter y getter para el espacio `age` creando primero genéricos con `setGeneric()`:

```
setGeneric("age", function(x) standardGeneric("age"))
setGeneric("age<-", function(x, value) standardGeneric("age<-"))
```

Y luego definiendo métodos con `setMethod()`:

```
setMethod("age", "Person", function(x) x@age)
setMethod("age<-", "Person", function(x, value) {
  x@age <- value
  x
})

age(john) <- 50
age(john)
#> [1] 50
```

Si está utilizando una clase S4 definida en un paquete, puede obtener ayuda con `class?Person`. Para obtener ayuda para un método, coloque `?` delante de una llamada (por ejemplo, `?age(john)`) y `?` usará la clase de los argumentos para averiguar qué archivo de ayuda necesita.

Finalmente, puede usar las funciones de sloop para identificar los objetos y genéricos de S4 que se encuentran en la naturaleza:

```
sloop::otype(john)
#> [1] "S4"
sloop::ftype(age)
#> [1] "S4"      "generic"
```

15.2.1. Ejercicios

1. `lubridate::period()` devuelve una clase S4. ¿Qué ranuras tiene? ¿Qué clase es cada ranura? ¿Qué accesorios proporciona?

2. ¿De qué otras formas puede encontrar ayuda para un método? Lea `?"?"` y resuma los detalles.

15.3. Clases

Para definir una clase S4, llama a `setClass()` con tres argumentos:

- La clase **nombre**. Por convención, los nombres de clase de S4 usan `UpperCamelCase`.
- Un vector de caracteres con nombre que describe los nombres y las clases de las **ranuras** (campos). Por ejemplo, una persona puede estar representada por un nombre de personaje y una edad numérica: `c(name = "personaje", age = "numérico")`. La pseudoclase `ANY` permite que una ranura acepte objetos de cualquier tipo.
- Un **prototipo**, una lista de valores predeterminados para cada ranura. Técnicamente, el prototipo es opcional[2], pero siempre debes proporcionarlo.

El siguiente código ilustra los tres argumentos mediante la creación de una clase `Person` con el carácter `name` y las ranuras numéricas `age`.

```
setClass("Person",
  slots = c(
    name = "character",
    age = "numeric"
  ),
  prototype = list(
    name = NA_character_,
    age = NA_real_
```

[2] `?setClass` recomienda que evite el argumento `prototype`, pero esto generalmente se considera un mal consejo.

```
  )
)

me <- new("Person", name = "Hadley")
str(me)
#> Formal class 'Person' [package ".GlobalEnv"] with 2 slots
#>   ..@ name: chr "Hadley"
#>   ..@ age : num NA
```

15.3.1. Herencia

Hay otro argumento importante para `setClass()`: `contains`. Esto especifica una clase (o clases) de las que heredar las ranuras y el comportamiento. Por ejemplo, podemos crear una clase `Employee` que herede de la clase `Person`, agregando un espacio adicional que describa su `boss`.

```
setClass("Employee",
  contains = "Person",
  slots = c(
    boss = "Person"
  ),
  prototype = list(
    boss = new("Person")
  )
)

str(new("Employee"))
#> Formal class 'Employee' [package ".GlobalEnv"] with 3 slots
#>   ..@ boss:Formal class 'Person' [package ".GlobalEnv"] with 2 slots
#>   .. .. ..@ name: chr NA
#>   .. .. ..@ age : num NA
```

```
#>   ..@ name: chr NA
#>   ..@ age : num NA
```

`setClass()` tiene otros 9 argumentos pero están en desuso o no se recomiendan.

15.3.2. Introspección

Para determinar de qué clases hereda un objeto, usa `is()`:

```
is(new("Person"))
#> [1] "Person"
is(new("Employee"))
#> [1] "Employee" "Person"
```

Para probar si un objeto hereda de una clase específica, usa el segundo argumento de `is()`:

```
is(john, "Person")
#> [1] TRUE
```

15.3.3. Redefinición

En la mayoría de los lenguajes de programación, la definición de clase ocurre en tiempo de compilación y la construcción de objetos ocurre más tarde, en tiempo de ejecución. En R, sin embargo, tanto la definición como la construcción ocurren en tiempo de ejecución. Cuando llamas a `setClass()`, estás registrando una definición de clase en una variable global (oculta). Al igual que con todas las funciones de modificación de estado, debe usar `setClass()` con cuidado. Es posible crear objetos no válidos si redefine una clase después de haber creado una instancia de un objeto:

```
setClass("A", slots = c(x = "numeric"))
a <- new("A", x = 10)

setClass("A", slots = c(a_different_slot = "numeric"))
a
#> An object of class "A"
#> Slot "a_different_slot":
#> Error in slot(object, what): no slot of name "a_different_slot" for this object of class
```

Esto puede causar confusión durante la creación interactiva de nuevas clases. (Las clases R6 tienen el mismo problema, como se describe en la Section 14.2.2.)

15.3.4. Ayudante

`new()` es un constructor de bajo nivel adecuado para que lo use usted, el desarrollador. Las clases orientadas al usuario siempre deben combinarse con un asistente fácil de usar. Un ayudante siempre debe:

- Tener el mismo nombre que la clase, p. `myclass()`.
- Tenga una interfaz de usuario cuidadosamente diseñada con valores predeterminados cuidadosamente seleccionados y conversiones útiles.
- Cree mensajes de error cuidadosamente elaborados y adaptados a un usuario final.
- Termine llamando a `methods::new()`.

La clase `Person` es tan simple que un ayudante es casi superfluo, pero podemos usarlo para definir claramente el contrato: `age` es opcional pero `name` es obligatorio. También forzaremos la edad a un doble para que el ayudante también funcione cuando se pasa un número entero.

```
Person <- function(name, age = NA) {
  age <- as.double(age)

  new("Person", name = name, age = age)
}

Person("Hadley")
#> An object of class "Person"
#> Slot "name":
#> [1] "Hadley"
#>
#> Slot "age":
#> [1] NA
```

15.3.5. Validador

El constructor verifica automáticamente que las ranuras tengan las clases correctas:

```
Person(mtcars)
#> Error in validObject(.Object): invalid class "Person" object: invalid obje
```

Deberá implementar verificaciones más complicadas (es decir, verificaciones que involucren longitudes o múltiples ranuras) usted mismo. Por ejemplo, es posible que queramos dejar en claro que la clase Person es una clase vectorial y puede almacenar datos sobre varias personas. Eso no está claro actualmente porque `@name` y `@age` pueden tener diferentes longitudes:

```
Person("Hadley", age = c(30, 37))
#> An object of class "Person"
#> Slot "name":
```

```
#> [1] "Hadley"
#>
#> Slot "age":
#> [1] 30 37
```

Para hacer cumplir estas restricciones adicionales, escribimos un validador con `setValidity()`. Toma una clase y una función que devuelve `TRUE` si la entrada es válida y, de lo contrario, devuelve un vector de caracteres que describe los problemas:

```
setValidity("Person", function(object) {
  if (length(object@name) != length(object@age)) {
    "@name and @age must be same length"
  } else {
    TRUE
  }
})
```

Ahora ya no podemos crear un objeto no válido:

```
Person("Hadley", age = c(30, 37))
#> Error in validObject(.Object): invalid class "Person" object: @name and @age must be same
```

NB: El método de validez solo es llamado automáticamente por `new()`, por lo que aún puede crear un objeto no válido modificándolo:

```
alex <- Person("Alex", age = 30)
alex@age <- 1:10
```

Puedes verificar explícitamente la validez tú mismo llamando a `validObject()`:

```
validObject(alex)
#> Error in validObject(alex): invalid class "Person" object: @name and @age
```

En la Section 15.4.4, usaremos `validObject()` para crear accesores que no pueden crear objetos no válidos.

15.3.6. Ejercicios

1. Extiende la clase Person con campos para que coincidan con `utils::person()`. Piense en qué ranuras necesitará, qué clase debe tener cada ranura y qué necesitará verificar en su método de validez.
2. ¿Qué sucede si define una nueva clase S4 que no tiene ranuras? (Sugerencia: lea acerca de las clases virtuales en `?setClass`).
3. Imagine que iba a volver a implementar factores, fechas y marcos de datos en S4. Esboce las llamadas `setClass()` que usaría para definir las clases. Piense en `slots` y `prototypes` apropiados.

15.4. Genéricos y métodos

El trabajo de un genérico es realizar el envío de métodos, es decir, encontrar la implementación específica para la combinación de clases pasadas al genérico. Aquí aprenderá a definir métodos y genéricos de S4; luego, en la siguiente sección, exploraremos con precisión cómo funciona el envío de métodos de S4.

Para crear un nuevo S4 genérico, llame a `setGeneric()` con una función que llame a `standardGeneric()`:

```
setGeneric("myGeneric", function(x) standardGeneric("myGeneric"))
```

Por convención, los nuevos genéricos de S4 deben usar `lowerCamelCase`.

Es una mala práctica usar `{}` en el genérico, ya que desencadena un caso especial que es más costoso y, en general, es mejor evitarlo.

```
# Don't do this!
setGeneric("myGeneric", function(x) {
  standardGeneric("myGeneric")
})
```

15.4.1. Firma

Al igual que `setClass()`, `setGeneric()` tiene muchos otros argumentos. Solo hay uno que debe conocer: `signature`. Esto le permite controlar los argumentos que se utilizan para el envío de métodos. Si no se proporciona `signature`, se utilizan todos los argumentos (excepto `...`). Ocasionalmente, es útil eliminar argumentos del envío. Esto le permite requerir que los métodos proporcionen argumentos como `verbose = TRUE` o `quiet = FALSE`, pero no toman parte en el envío.

```
setGeneric("myGeneric",
  function(x, ..., verbose = TRUE) standardGeneric("myGeneric"),
  signature = "x"
)
```

15.4.2. Métodos

Un genérico no es útil sin algunos métodos, y en S4 se definen métodos con `setMethod()`. Hay tres argumentos importantes: el nombre del genérico, el nombre de la clase y el método en sí.

```
setMethod("myGeneric", "Person", function(x) {
  # method implementation
})
```

Más formalmente, el segundo argumento de `setMethod()` se llama **signature**. En S4, a diferencia de S3, la firma puede incluir múltiples argumentos. Esto hace que el envío de métodos en S4 sea sustancialmente más complicado, pero evita tener que implementar el envío doble como un caso especial. Hablaremos más sobre el envío múltiple en la siguiente sección. `setMethod()` tiene otros argumentos, pero nunca debes usarlos.

Para listar todos los métodos que pertenecen a un genérico, o que están asociados con una clase, use `methods("generic")` o `methods(class = "class")`; para encontrar la implementación de un método específico, use `selectMethod("generic", "class")`.

15.4.3. Mostrar método

El método S4 más comúnmente definido que controla la impresión es `show()`, que controla cómo aparece el objeto cuando se imprime. Para definir un método para un genérico existente, primero debe determinar los argumentos. Puede obtenerlos de la documentación o mirando los `args()` del genérico:

```
args(getGeneric("show"))
#> function (object)
#> NULL
```

Nuestro método show necesita tener un solo argumento `object`:

```
setMethod("show", "Person", function(object) {
  cat(is(object)[[1]], "\n",
      "  Name: ", object@name, "\n",
      "  Age:  ", object@age, "\n",
      sep = ""
  )
})
john
#> Person
#>   Name: John Smith
#>   Age:  50
```

15.4.4. Accesorios

Las ranuras deben considerarse un detalle de implementación interna: pueden cambiar sin previo aviso y el código de usuario debe evitar acceder a ellas directamente. En su lugar, todas las ranuras accesibles para el usuario deben ir acompañadas de un par de **accesorios**. Si la ranura es única para la clase, esto puede ser solo una función:

```
person_name <- function(x) x@name
```

Sin embargo, por lo general, definirá un genérico para que varias clases puedan usar la misma interfaz:

```
setGeneric("name", function(x) standardGeneric("name"))
setMethod("name", "Person", function(x) x@name)

name(john)
#> [1] "John Smith"
```

Si la ranura también se puede escribir, debe proporcionar una función de establecimiento. Siempre debe incluir `validObject()` en el setter para evitar que el usuario cree objetos no válidos.

```
setGeneric("name<-", function(x, value) standardGeneric("name<-"))
setMethod("name<-", "Person", function(x, value) {
  x@name <- value
  validObject(x)
  x
})

name(john) <- "Jon Smythe"
name(john)
#> [1] "Jon Smythe"

name(john) <- letters
#> Error in validObject(x): invalid class "Person" object: @name and @age mus
```

(Si la notación `name<-` no le resulta familiar, revise la Section 6.8.)

15.4.5. Ejercicios

1. Agregue accesores `age()` para la clase `Person`.
2. En la definición de genérico, ¿por qué es necesario repetir dos veces el nombre del genérico?
3. ¿Por qué el método `show()` definido en la Section 15.4.3 usa `is(object)[[1]]`? (Sugerencia: intente imprimir la subclase de empleado).
4. ¿Qué pasa si defines un método con nombres de argumentos diferentes al genérico?

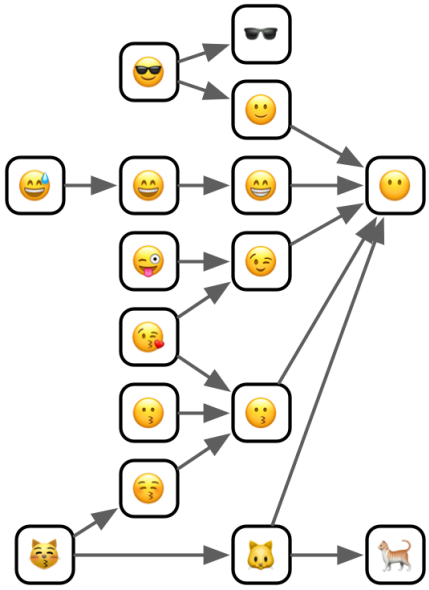
15.5. Método de envío

El envío de S4 es complicado porque S4 tiene dos características importantes:

- Herencia múltiple, es decir, una clase puede tener múltiples padres,
- Envío múltiple, es decir, un genérico puede usar múltiples argumentos para elegir un método.

Estas características hacen que S4 sea muy potente, pero también pueden dificultar la comprensión de qué método se seleccionará para una determinada combinación de entradas. En la práctica, mantenga el envío de métodos lo más simple posible evitando la herencia múltiple y reservando el envío múltiple solo cuando sea absolutamente necesario.

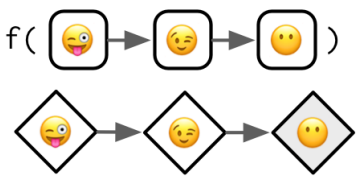
Pero es importante describir los detalles completos, por lo que aquí comenzaremos de manera simple con herencia única y despacho único, y avanzaremos hasta los casos más complicados. Para ilustrar las ideas sin atascarse en los detalles, usaremos un **gráfico de clase** imaginario basado en emoji:



Emoji nos da nombres de clase muy compactos que evocan las relaciones entre las clases. Debería ser sencillo recordar que 😜 hereda de 😉 que hereda de 😶, y que 😎 hereda tanto de 🕶️ como de 🙂.

15.5.1. Envío único

Comencemos con el caso más simple: una función genérica que se distribuye en una sola clase con un solo padre. El método de envío aquí es simple, por lo que es un buen lugar para definir las convenciones gráficas que usaremos para los casos más complejos.



Hay dos partes en este diagrama:

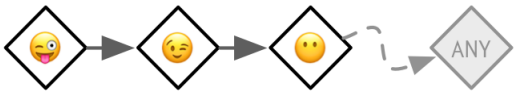
- La parte superior, `f(...)`, define el alcance del diagrama. Aquí tenemos un genérico con un argumento, que tiene una jerarquía de clases de tres niveles de profundidad.

- La parte inferior es el **gráfico de métodos** y muestra todos los métodos posibles que podrían definirse. Los métodos que existen, es decir, que se han definido con `setMethod()`, tienen un fondo gris.

Para encontrar el método que se llama, comience con la clase más específica de los argumentos reales, luego siga las flechas hasta que encuentre un método que exista. Por ejemplo, si llamaste a la función con un objeto de la clase 😉, seguirías la flecha hacia la derecha para encontrar el método definido para la clase más general 😶. Si no se encuentra ningún método, el envío del método ha fallado y se genera un error. En la práctica, esto significa que siempre debe definir métodos definidos para los nodos terminales, es decir, los del extremo derecho.

Hay dos **pseudoclases** para las que puede definir métodos. Estas se denominan pseudoclases porque en realidad no existen, pero le permiten definir comportamientos útiles. La primera pseudoclase es `ANY` que coincide con cualquier clase[3]. Por razones técnicas que veremos más adelante, el enlace al método `ANY` es más largo que los enlaces entre las otras clases:

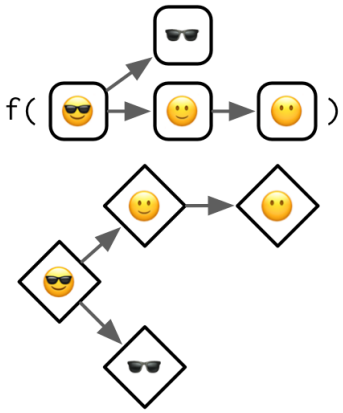
[3]La pseudoclase `ANY` de S4 desempeña el mismo papel que la pseudoclase `default` de S3.



La segunda pseudoclase es `MISSING`. Si define un método para esta pseudoclase, coincidirá siempre que falte el argumento. No es útil para envío único, pero es importante para funciones como `+` y `-` que usan envío doble y se comportan de manera diferente dependiendo de si tienen uno o dos argumentos.

15.5.2. Herencia múltiple

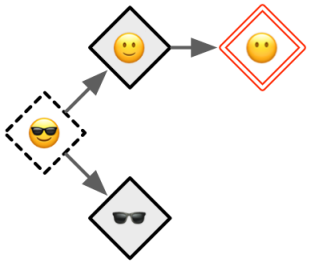
Las cosas se complican más cuando la clase tiene varios padres.



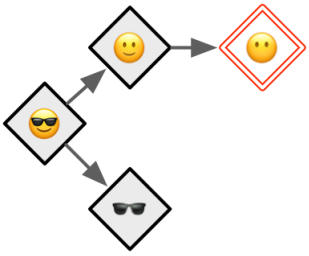
El proceso básico sigue siendo el mismo: comienza desde la clase real suministrada al genérico, luego sigue las flechas hasta encontrar un método definido. El problema es que ahora hay varias flechas a seguir, por lo que es posible que encuentre varios métodos. Si eso sucede, elige el método más cercano, es decir, requiere viajar con la menor cantidad de flechas.

NB: Si bien el gráfico de métodos es una poderosa metáfora para comprender el envío de métodos, implementarlo de esta manera sería bastante ineficiente, por lo que el enfoque real que usa S4 es algo diferente. Puede leer los detalles en `?Methods_Details`.

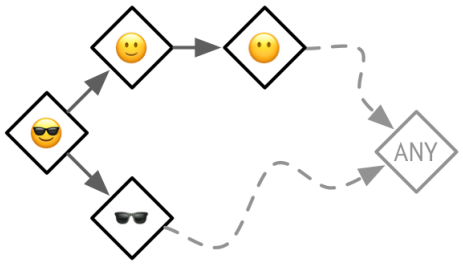
¿Qué sucede si los métodos están a la misma distancia? Por ejemplo, imagina que hemos definido métodos para 🕶 y 🙂, y llamamos al genérico 😎. Tenga en cuenta que no se puede encontrar ningún método para la clase 😶, que resaltaré con un doble contorno rojo.



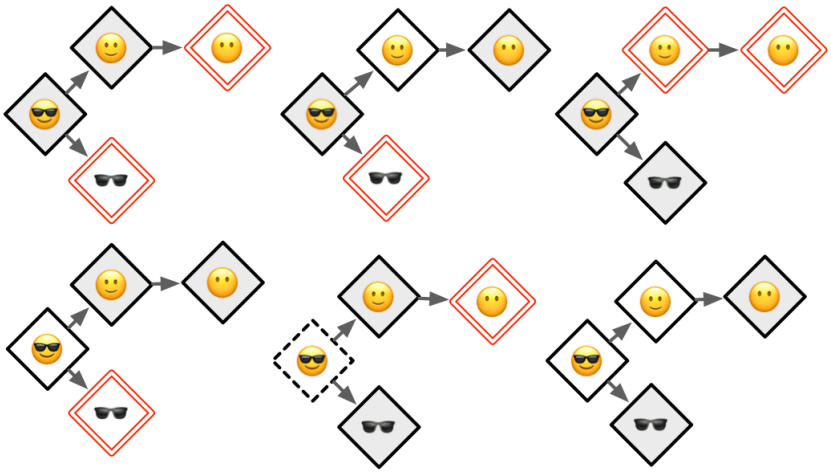
Esto se llama método **ambiguo**, y en los diagramas lo ilustraré con un borde de puntos gruesos. Cuando esto sucede en R, recibirá una advertencia y se elegirá el método para la clase que aparece antes en el alfabeto (esto es efectivamente aleatorio y no se debe confiar en él). Cuando descubra una ambigüedad, siempre debe resolverla proporcionando un método más preciso:



El método alternativo `ANY` aún existe, pero las reglas son un poco más complejas. Como lo indican las líneas punteadas onduladas, el método `ANY` siempre se considera más lejano que un método para una clase real. Esto significa que nunca contribuirá a la ambigüedad.

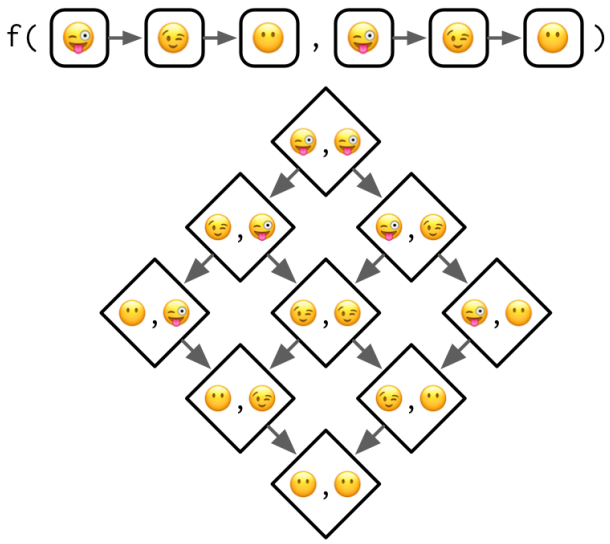


Con herencias múltiples, es difícil evitar simultáneamente la ambigüedad, asegurarse de que cada método de terminal tenga una implementación y minimizar la cantidad de métodos definidos (para beneficiarse de OOP). Por ejemplo, de las seis formas de definir solo dos métodos para esta llamada, solo una está libre de problemas. Por esta razón, recomiendo utilizar la herencia múltiple con sumo cuidado: deberá pensar detenidamente en el gráfico del método y planificar en consecuencia.



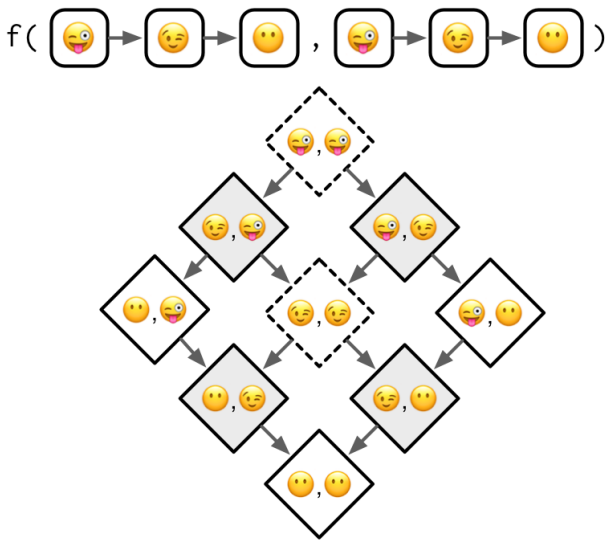
15.5.3. Envío múltiple

Una vez que comprenda la herencia múltiple, comprender el envío múltiple es sencillo. Sigue varias flechas de la misma manera que antes, pero ahora cada método se especifica mediante dos clases (separadas por una coma).



No voy a mostrar ejemplos de despacho en más de dos argumentos, pero puede seguir los principios básicos para generar sus propios gráficos de métodos.

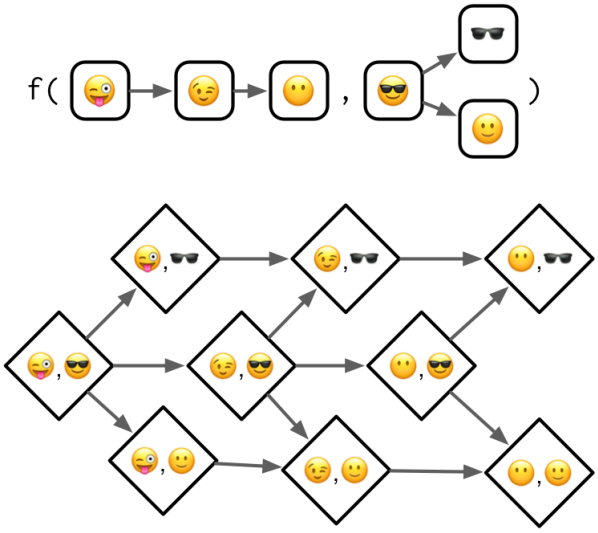
La principal diferencia entre la herencia múltiple y el envío múltiple es que hay muchas más flechas a seguir. El siguiente diagrama muestra cuatro métodos definidos que producen dos casos ambiguos:



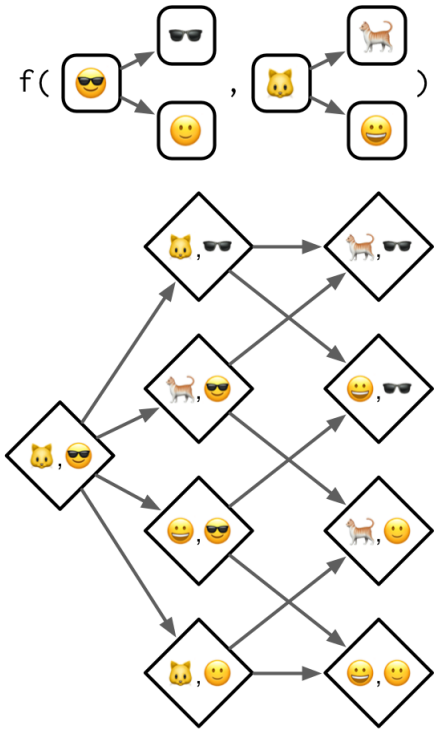
El envío múltiple tiende a ser menos complicado para trabajar que la herencia múltiple porque generalmente hay menos combinaciones de clases de terminales. En este ejemplo, solo hay uno. Eso significa que, como mínimo, puede definir un solo método y tener un comportamiento predeterminado para todas las entradas.

15.5.4. Despacho múltiple y herencia múltiple

Por supuesto, puede combinar envío múltiple con herencia múltiple:



Un caso aún más complicado se despacha en dos clases, las cuales tienen herencia múltiple:



A medida que el gráfico del método se vuelve más y más complicado, se vuelve más y más difícil predecir qué método se llamará dada una combinación de entradas, y se vuelve más y más difícil asegurarse de que no se ha introducido ambigüedad. Si tiene que dibujar diagramas para averiguar qué método se llamará realmente, es una fuerte indicación de que debe volver atrás y simplificar su diseño.

15.5.5. Ejercicios

1. Dibuje el gráfico del método para `f(`😅, 😪`)`.

2. Dibuje el gráfico del método para `f(😃, 😉, 😙)`.

3. Tome el último ejemplo que muestra envío múltiple sobre dos clases que usan herencia múltiple. ¿Qué sucede si define un método para todas las clases de terminal? ¿Por qué el método de envío no nos ahorra mucho trabajo aquí?

15.6. S4 y S3

Al escribir código S4, a menudo necesitará interactuar con clases y genéricos existentes de S3. Esta sección describe cómo las clases, los métodos y los genéricos de S4 interactúan con el código existente.

15.6.1. Clases

En `slots` y `contains` puede usar clases S4, clases S3 o la clase implícita (Section 13.7.1) de un tipo base. Para usar una clase S3, primero debe registrarla con `setOldClass()`. Llame a esta función una vez para cada clase de S3, dándole el atributo de clase. Por ejemplo, la base R ya proporciona las siguientes definiciones:

```
setOldClass("data.frame")
setOldClass(c("ordered", "factor"))
setOldClass(c("glm", "lm"))
```

Sin embargo, generalmente es mejor ser más específico y proporcionar una definición completa de S4 con `slots` y un `prototype`:

```
setClass("factor",
  contains = "integer",
  slots = c(
```

```
    levels = "character"
  ),
  prototype = structure(
    integer(),
    levels = character()
  )
)
setOldClass("factor", S4Class = "factor")
```

Por lo general, estas definiciones las debe proporcionar el creador de la clase S3. Si intenta crear una clase S4 sobre una clase S3 proporcionada por un paquete, debe solicitar que el mantenedor del paquete agregue esta llamada a su paquete, en lugar de agregarla a su propio código.

Si un objeto S4 hereda de una clase S3 o un tipo base, tendrá una ranura virtual especial llamada `.Data`. Esto contiene el tipo base subyacente o el objeto S3:

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

Es posible definir métodos de S3 para genéricos de S4 y métodos de S4 para genéricos de S3 (siempre que haya llamado a `setOldClass()`). Sin embargo, es más complicado de lo que parece a primera vista, así que asegúrese de leer detenidamente `?Methods_for_S3`.

15.6.2. Genéricos

Además de crear un nuevo genérico desde cero, también es posible convertir un genérico S3 existente en un genérico S4:

```
setGeneric("mean")
```

En este caso, la función existente se convierte en el método predeterminado (`ANY`):

```
selectMethod("mean", "ANY")
#> Method Definition (Class "derivedDefaultMethod"):
#>
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x55f20668be78>
#> <environment: namespace:base>
#>
#> Signatures:
#>         x
#> target  "ANY"
#> defined "ANY"
```

NB: `setMethod()` llamará automáticamente a `setGeneric()` si el primer argumento aún no es genérico, lo que le permite convertir cualquier función existente en un genérico S4. Está bien convertir un S3 genérico existente a S4, pero debe evitar convertir funciones regulares a genéricos S4 en paquetes porque eso requiere una coordinación cuidadosa si lo hacen varios paquetes.

15.6.3. Ejercicios

1. ¿Cómo sería una definición completa de `setOldClass()` para un factor ordenado (es decir, agregar `slots` y `prototype` de la definición anterior)?
2. Defina un método `length` para la clase `Person`.

16. Compensaciones

16.1. Introducción

Ahora conoce los tres conjuntos de herramientas OOP más importantes disponibles en R. Ahora que comprende su funcionamiento básico y los principios que los sustentan, podemos comenzar a comparar y contrastar los sistemas para comprender sus fortalezas y debilidades. Esto le ayudará a elegir el sistema que tiene más probabilidades de resolver nuevos problemas.

En general, al elegir un sistema OO, le recomiendo que utilice S3 de forma predeterminada. S3 es simple y se usa ampliamente en base R y CRAN. Si bien está lejos de ser perfecto, sus idiosincrasias se comprenden bien y existen enfoques conocidos para superar la mayoría de las deficiencias. Si tiene experiencia previa en programación, es probable que se incline hacia R6, porque le resultará familiar. Creo que deberías resistirte a esta tendencia por dos razones. En primer lugar, si usa R6, es muy fácil crear una API no idiomática que se sentirá muy extraña para los usuarios nativos de R y tendrá puntos débiles sorprendentes debido a la semántica de referencia. En segundo lugar, si se apega a R6, perderá el aprendizaje de una nueva forma de pensar sobre OOP que le brinda un nuevo conjunto de herramientas para resolver problemas.

Estructura

- La Section 16.2 compara S3 y S4. En resumen, S4 es más formal y tiende a requerir una planificación más anticipada. Eso lo hace más adecuado para grandes proyectos desarrollados por equipos, no individualmente.
- La Section 16.3 compara S3 y R6. Esta sección es bastante larga porque estos dos sistemas son fundamentalmente diferentes y hay una serie de compensaciones que debe tener en cuenta.

Requisitos previos

Debe estar familiarizado con S3, S4 y R6, como se explicó en los tres capítulos anteriores.

16.2. S4 contra S3

Una vez que haya dominado S3, S4 no es demasiado difícil de entender: las ideas subyacentes son las mismas, S4 es simplemente más formal, más estricto y más detallado. El rigor y la formalidad de S4 lo hacen ideal para equipos grandes. Dado que el propio sistema proporciona más estructura, hay menos necesidad de convenciones y los nuevos contribuyentes no necesitan tanta formación. S4 tiende a requerir un diseño más inicial que S3, y es más probable que esta inversión rinda frutos en proyectos más grandes donde hay más recursos disponibles.

Un gran esfuerzo de equipo donde S4 se usa con buenos resultados es Bioconductor. Bioconductor es similar a CRAN: es una forma de compartir paquetes entre una audiencia más amplia. Bioconductor es más pequeño que CRAN (~1,300 versus ~10,000 paquetes, julio de 2017) y los paquetes tienden a estar más estrechamente integrados debido al dominio compartido y porque Bioconductor tiene un proceso de revisión más estricto. No

se requieren paquetes de bioconductores para usar S4, pero la mayoría lo hará porque las estructuras de datos clave (por ejemplo, SummarizedExperiment, IRanges, DNAStringSet) se construyen usando S4.

S4 también es una buena opción para sistemas complejos de objetos interrelacionados, y es posible minimizar la duplicación de código mediante la implementación cuidadosa de métodos. El mejor ejemplo de tal sistema es el paquete Matrix (Bates and Maechler 2018). Está diseñado para almacenar y calcular de manera eficiente con muchos tipos diferentes de matrices densas y dispersas. A partir de la versión 1.7.0, define clases 108, funciones genéricas 23 y métodos 1780, y para darle una idea de la complejidad, se muestra un pequeño subconjunto del gráfico de clase en Figure 16.1.

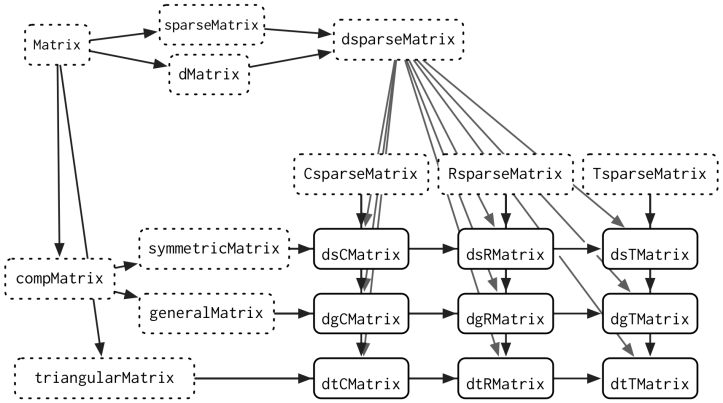


Figure 16.1.: Un pequeño subconjunto del gráfico de la clase Matrix que muestra la herencia de matrices dispersas. Cada clase concreta hereda de dos padres virtuales: uno que describe cómo se almacenan los datos (C = orientado a columnas, R = orientado a filas, T = etiquetado) y otro que describe cualquier restricción en la matriz (s = simétrico, t = triángulo, g = generales).

Este dominio es una buena opción para S4 porque a menudo hay ata-

jos computacionales para combinaciones específicas de matrices dispersas. S4 facilita proporcionar un método general que funcione para todas las entradas y luego proporcionar métodos más especializados donde las entradas permiten una implementación más eficiente. Esto requiere una planificación cuidadosa para evitar la ambigüedad en el envío de métodos, pero la planificación compensa con un mayor rendimiento.

El mayor desafío para usar S4 es la combinación de una mayor complejidad y la ausencia de una única fuente de documentación. S4 es un sistema complejo y su uso eficaz en la práctica puede resultar complicado. Esto no sería un gran problema si la documentación de S4 no estuviera dispersa en la documentación, los libros y los sitios web de R. S4 necesita un tratamiento de longitud de libro, pero ese libro (todavía) no existe. (La documentación para S3 no es mejor, pero la falta es menos dolorosa porque S3 es mucho más simple).

16.3. R6 contra S3

R6 es un sistema OO profundamente diferente de S3 y S4 porque se basa en objetos encapsulados, en lugar de funciones genéricas. Además, los objetos R6 tienen semántica de referencia, lo que significa que se pueden modificar en su lugar. Estas dos grandes diferencias tienen una serie de consecuencias no obvias que exploraremos aquí:

- Un genérico es una función regular, por lo que vive en el espacio de nombres global. Un método R6 pertenece a un objeto, por lo que vive en un espacio de nombres local. Esto influye en cómo pensamos acerca de nombrar.

- La semántica de referencia de R6 permite que los métodos devuelvan un valor y modifiquen un objeto simultáneamente. Esto resuelve un doloroso problema llamado “estado de subprocesamiento”.

- Invocas un método R6 usando `$`, que es un operador infijo. Si configura sus métodos correctamente, puede usar cadenas de llamadas a métodos como una alternativa a la canalización.

Estas son compensaciones generales entre OOP funcional y encapsulado, por lo que también sirven como una discusión sobre el diseño del sistema en R versus Python.

16.3.1. Espacio de nombres

Una diferencia no obvia entre S3 y R6 es el espacio en el que se encuentran los métodos:

- Las funciones genéricas son globales: todos los paquetes comparten el mismo espacio de nombres.
- Los métodos encapsulados son locales: los métodos están vinculados a un solo objeto.

La ventaja de un espacio de nombres global es que varios paquetes pueden usar los mismos verbos para trabajar con diferentes tipos de objetos. Las funciones genéricas proporcionan una API uniforme que facilita la realización de acciones típicas con un nuevo objeto porque existen fuertes convenciones de nomenclatura. Esto funciona bien para el análisis de datos porque a menudo desea hacer lo mismo con diferentes tipos de objetos. En particular, esta es una de las razones por las que el sistema de modelado de R es tan útil: independientemente de dónde se haya implementado el modelo, siempre se trabaja con él usando el mismo conjunto de herramientas (`summary()`, `predict()`, …).

La desventaja de un espacio de nombres global es que lo obliga a pensar más profundamente sobre la asignación de nombres. Desea evitar múltiples genéricos con el mismo nombre en diferentes paquetes porque requiere que el usuario escriba `::` con frecuencia. Esto puede ser difícil porque los

nombres de las funciones suelen ser verbos en inglés y los verbos suelen tener varios significados. Toma `plot()` por ejemplo:

```
plot(data)       # plot some data
plot(bank_heist) # plot a crime
plot(land)       # create a new plot of land
plot(movie)      # extract plot of a movie
```

En general, debe evitar los métodos que son homónimos del genérico original y, en su lugar, definir un nuevo genérico.

Este problema no ocurre con los métodos R6 porque están en el ámbito del objeto. El siguiente código está bien, porque no implica que el método de trazado de dos objetos R6 diferentes tenga el mismo significado:

```
data$plot()
bank_heist$plot()
land$plot()
movie$plot()
```

Estas consideraciones también se aplican a los argumentos a la genérica. Los genéricos de S3 deben tener los mismos argumentos centrales, lo que significa que generalmente tienen nombres no específicos como `x` o `.data`. Los genéricos de S3 generalmente necesitan `...` para pasar argumentos adicionales a los métodos, pero esto tiene la desventaja de que los nombres de los argumentos mal escritos no generarán un error. En comparación, los métodos R6 pueden variar más ampliamente y usar nombres de argumentos más específicos y sugerentes.

Una ventaja secundaria del espacio de nombres local es que crear un método R6 es muy económico. La mayoría de los lenguajes OO encapsulados lo alientan a crear muchos métodos pequeños, cada uno de los cuales hace una cosa bien con un nombre evocador. Crear un nuevo método S3 es más costoso, porque es posible que también deba crear uno genérico y

pensar en los problemas de nombres descritos anteriormente. Eso significa que el consejo de crear muchos métodos pequeños no se aplica a S3. Todavía es una buena idea dividir el código en fragmentos pequeños y fáciles de entender, pero por lo general deberían ser solo funciones regulares, no métodos.

16.3.2. Estado de enhebrado

Un desafío de programar con S3 es cuando desea devolver un valor y modificar el objeto. Esto viola nuestra pauta de que se debe llamar a una función por su valor de retorno o por sus efectos secundarios, pero es necesario en algunos casos.

Por ejemplo, imagina que quieres crear una **pila** de objetos. Una pila tiene dos métodos principales:

- `push()` agrega un nuevo objeto a la parte superior de la pila.
- `pop()` devuelve el valor superior y lo elimina de la pila.

La implementación del constructor y el método `push()` es sencilla. Una pila contiene una lista de elementos, y empujar un objeto a la pila simplemente se agrega a esta lista.

```
new_stack <- function(items = list()) {
  structure(list(items = items), class = "stack")
}

push <- function(x, y) {
  x$items <- c(x$items, list(y))
  x
}
```

(No he creado un método real para `push()` porque hacerlo genérico solo haría que este ejemplo fuera más complicado sin ningún beneficio real).

Implementar `pop()` es más desafiante porque tiene que devolver un valor (el objeto en la parte superior de la pila) y tener un efecto secundario (eliminar ese objeto de esa parte superior). Como no podemos modificar el objeto de entrada en S3, debemos devolver dos cosas: el valor y el objeto actualizado.

```
pop <- function(x) {
  n <- length(x$items)

  item <- x$items[[n]]
  x$items <- x$items[-n]

  list(item = item, x = x)
}
```

Esto conduce a un uso bastante incómodo:

```
s <- new_stack()
s <- push(s, 10)
s <- push(s, 20)

out <- pop(s)
out$item
#> [1] 20
s <- out$x
s
#> $items
#> $items[[1]]
#> [1] 10
#>
#>
#> attr(,"class")
#> [1] "stack"
```

Este problema se conoce como **estado de subprocesamiento** o **programación del acumulador**, porque no importa qué tan profundamente se llame a `pop()`, debe enhebrar el objeto de pila modificado hasta donde vive.

Una forma en que otros lenguajes de FP enfrentan este desafío es proporcionar un operador de **asignación múltiple** (o enlace de desestructuración) que le permite asignar múltiples valores en un solo paso. El paquete zeallot (Teetor 2018) proporciona asignaciones múltiples para R con `%<-%`. Esto hace que el código sea más elegante, pero no resuelve el problema clave:

```
library(zeallot)

c(value, s) %<-% pop(s)
value
#> [1] 10
```

Una implementación R6 de una pila es más simple porque `$pop()` puede modificar el objeto en su lugar y devolver solo el valor más alto:

```
Stack <- R6::R6Class("Stack", list(
  items = list(),
  push = function(x) {
    self$items <- c(self$items, x)
    invisible(self)
  },
  pop = function() {
    item <- self$items[[self$length()]]
    self$items <- self$items[-self$length()]
    item
  },
  length = function() {
    length(self$items)
```

```
  }
))
```

Esto conduce a un código más natural:

```
s <- Stack$new()
s$push(10)
s$push(20)
s$pop()
#> [1] 20
```

Encontré un ejemplo de la vida real del estado de subprocesamiento en las escalas ggplot2. Las escalas son complejas porque necesitan combinar datos en cada faceta y cada capa. Originalmente usé clases S3, pero requería pasar datos de escala hacia y desde muchas funciones. Cambiar a R6 simplificó sustancialmente el código. Sin embargo, también introdujo algunos problemas porque olvidé llamar a `$clone()` al modificar un gráfico. Esto permitió que las parcelas independientes compartieran los mismos datos de escala, creando un error sutil que era difícil de rastrear.

16.3.3. Encadenamiento de métodos

La canalización, `|>`, es útil porque proporciona un operador infijo que facilita la composición de funciones de izquierda a derecha. Curiosamente, la tubería no es tan importante para los objetos R6 porque ya usan un operador infijo: `$`. Esto permite al usuario encadenar varias llamadas a métodos en una sola expresión, una técnica conocida como **encadenamiento de métodos**:

```
s <- Stack$new()
s$
  push(10)$
```

```
  push(20)$
  pop()
#> [1] 20
```

Esta técnica se usa comúnmente en otros lenguajes de programación, como Python y JavaScript, y es posible con una convención: cualquier método R6 que se llame principalmente por sus efectos secundarios (generalmente modificando el objeto) debe devolver `invisible(self)`.

La principal ventaja del encadenamiento de métodos es que puede obtener un autocompletado útil; la desventaja principal es que solo el creador de la clase puede agregar nuevos métodos (y no hay forma de usar el envío múltiple).

Part IV.

Metaprogramación

Introducción

Una de las cosas más intrigantes de R es su capacidad para realizar **metaprogramación**. Esta es la idea de que el código son datos que se pueden inspeccionar y modificar mediante programación. Esta es una idea poderosa; uno que influye profundamente en mucho código R. En el nivel más básico, le permite hacer cosas como escribir `library("purrr")` en lugar de `library("purrr")` y habilitar `plot(x, sin(x))` para etiquetar automáticamente los ejes con `x` y `sin(x)`. En un nivel más profundo, te permite hacer cosas como usar `y ~ x1 + x2` para representar un modelo que predice el valor de `y` a partir de `x1` y `x2`, para traducir `subset(df, x == y)` en `df[df$x == df$y, , drop = FALSE]`, y usar `dplyr::filter(db, is.na(x))` para generar el SQL `WHERE x IS NULL` cuando `db` es una tabla de base de datos remota.

Estrechamente relacionado con la metaprogramación está la **evaluación no estándar**, NSE para abreviar. Este término, que se usa comúnmente para describir el comportamiento de las funciones R, es problemático de dos maneras. En primer lugar, NSE es en realidad una propiedad del argumento (o argumentos) de una función, por lo que hablar de funciones NSE es un poco descuidado. En segundo lugar, es confuso definir algo por lo que no es (estándar), por lo que en este libro presentaré un vocabulario más preciso.

Específicamente, este libro se enfoca en la evaluación ordenada (a veces llamada evaluación ordenada para abreviar). La evaluación ordenada se implementa en el paquete rlang (Henry and Wickham 2018b), y usaré rlang extensamente en estos capítulos. Esto le permitirá concentrarse en las grandes ideas, sin distraerse con las peculiaridades de la implementación

que surgen de la historia de R. Después de presentar cada gran idea con rlang, regresaré para hablar sobre cómo se expresan esas ideas en la base R. Este enfoque puede parecer atrasado para algunos, pero es como aprender a conducir usando una transmisión automática en lugar de una palanca. cambio: le permite concentrarse en el panorama general antes de tener que aprender los detalles. Este libro se centra en el lado teórico de la evaluación ordenada, para que pueda comprender completamente cómo funciona desde cero. Si está buscando una introducción más práctica, le recomiendo el libro de evaluación ordenado en https://tidyeval.tidyverse.org[1].

Aprenderá sobre la metaprogramación y la evaluación ordenada en los siguientes cinco capítulos:

1. El Chapter 17 brinda una descripción de alto nivel de toda la historia de la metaprogramación, aprendiendo brevemente sobre todos los componentes principales y cómo encajan para formar un todo cohesivo.

2. El Chapter 18 muestra que todo el código R se puede describir como un árbol. Aprenderá cómo visualizar estos árboles, cómo las reglas de la gramática de R convierten secuencias lineales de caracteres en estos árboles y cómo usar funciones recursivas para trabajar con árboles de código.

3. El Chapter 19 presenta herramientas de rlang que puede usar para capturar (citar) argumentos de función no evaluados. También aprenderá sobre la cuasicomilla, que proporciona un conjunto de técnicas para eliminar las comillas de entrada para que sea posible generar fácilmente nuevos árboles a partir de fragmentos de código.

4. El Chapter 20 pasa a evaluar el código capturado. Aquí aprenderá sobre una estructura de datos importante, el **quosure**, que garantiza una evaluación correcta al capturar tanto el código para evaluar

[1]Mientras escribo este capítulo, el ordenado libro de evaluación es todavía un trabajo en progreso, pero para cuando lo lea, espero que esté terminado.

como el entorno en el que se evalúa. Este capítulo le mostrará cómo juntar todas las piezas para comprender cómo funciona NSE en base R y cómo escribir funciones que funcionen como `subset()`.

5. El Chapter 21 termina combinando entornos de primera clase, alcance léxico y metaprogramación para traducir el código R a otros lenguajes, a saber, HTML y LaTeX.

17. Panorama general

17.1. Introducción

La metaprogramación es el tema más difícil de este libro porque reúne muchos temas que antes no estaban relacionados y lo obliga a lidiar con problemas en los que probablemente no había pensado antes. También necesitarás aprender mucho vocabulario nuevo, y al principio parecerá que cada término nuevo está definido por otros tres términos de los que no has oído hablar. Incluso si es un programador experimentado en otro idioma, es poco probable que sus habilidades existentes sean de mucha ayuda, ya que pocos lenguajes populares modernos exponen el nivel de metaprogramación que proporciona R. Así que no se sorprenda si se siente frustrado o confundido al principio; ¡Esta es una parte natural del proceso que le sucede a todos!

Pero creo que ahora es más fácil aprender metaprogramación que nunca. En los últimos años, la teoría y la práctica han madurado sustancialmente, brindando una base sólida junto con herramientas que le permiten resolver problemas comunes. En este capítulo, obtendrá una visión general de todas las piezas principales y cómo encajan entre sí.

Estructura

Cada sección de este capítulo presenta una gran idea nueva:

- La Section 17.2 muestra que el código son datos y le enseña cómo crear y modificar expresiones mediante la captura de código.
- La Section 17.3 describe la estructura del código en forma de árbol, llamada árbol de sintaxis abstracta.
- La Section 17.4 muestra cómo crear nuevas expresiones programáticamente.
- La Section 17.5) muestra cómo ejecutar expresiones evaluándolas en un entorno.
- La Section 17.6 ilustra cómo personalizar la evaluación proporcionando funciones personalizadas en un nuevo entorno.
- La Section 17.7 extiende esa personalización a las máscaras de datos, que desdibujan la línea entre los entornos y los data frames.
- La Section 17.8 introduce una nueva estructura de datos llamada quosure que hace que todo esto sea más simple y correcto.

Requisitos previos

Este capítulo presenta las grandes ideas usando rlang; aprenderá los equivalentes básicos en capítulos posteriores. También usaremos el paquete lobstr para explorar la estructura de árbol del código.

```
library(rlang)
library(lobstr)
```

Asegúrese de que también está familiarizado con las estructuras de datos del entorno (Section 7.2) y del data frame (Section 3.6).

17.2. El código es datos

La primera gran idea es que el código es información: puede capturar código y calcularlo como puede hacerlo con cualquier otro tipo de información. La primera forma de capturar código es con `rlang::expr()`. Puedes pensar en `expr()` como si devolviera exactamente lo que pasas:

```
expr(mean(x, na.rm = TRUE))
#> mean(x, na.rm = TRUE)
expr(10 + 100 + 1000)
#> 10 + 100 + 1000
```

Más formalmente, el código capturado se llama **expresión**. Una expresión no es un único tipo de objeto, sino un término colectivo para cualquiera de los cuatro tipos (llamada, símbolo, constante o lista de pares), sobre los que aprenderá más en el Chapter 18.

`expr()` le permite capturar el código que ha escrito. Necesita una herramienta diferente para capturar el código pasado a una función porque `expr()` no funciona:

```
capture_it <- function(x) {
  expr(x)
}
capture_it(a + b + c)
#> x
```

Aquí debe usar una función diseñada específicamente para capturar la entrada del usuario en un argumento de función: `enexpr()`. Piensa en "en" en el contexto de "enriquecer": `enexpr()` toma un argumento mal evaluado y lo convierte en una expresión:

```
capture_it <- function(x) {
  enexpr(x)
}
capture_it(a + b + c)
#> a + b + c
```

Como `capture_it()` usa `enexpr()`, decimos que cita automáticamente su primer argumento. Aprenderá más sobre este término en la Section 19.2.1.

Una vez que haya capturado una expresión, puede inspeccionarla y modificarla. Las expresiones complejas se comportan como listas. Eso significa que puedes modificarlos usando `[[` y `$`:

```
f <- expr(f(x = 1, y = 2))

# Agregar un nuevo argumento
f$z <- 3
f
#> f(x = 1, y = 2, z = 3)

# Or eliminar un argumento
f[[2]] <- NULL
f
#> f(y = 2, z = 3)
```

El primer elemento de la llamada es la función a llamar, lo que significa que el primer argumento está en la segunda posición. Conocerá los detalles completos en la Section 18.3.

17.3. El código es un árbol

Para realizar manipulaciones más complejas con expresiones, debe comprender completamente su estructura. Detrás de escena, casi todos los lenguajes de programación representan el código como un árbol, a menudo llamado **árbol de sintaxis abstracta**, o AST para abreviar. R es inusual en el sentido de que realmente puede inspeccionar y manipular este árbol.

Una herramienta muy conveniente para comprender la estructura en forma de árbol es `lobstr::ast()`. Dado algo de código, esta función muestra la estructura de árbol subyacente. Las llamadas a funciones forman las ramas del árbol y se muestran mediante rectángulos. Las hojas del árbol son símbolos (como `a`) y constantes (como `"b"`).

```
lobstr::ast(f(a, "b"))
#>  f
#>  a
#>  "b"
```

Las llamadas a funciones anidadas crean árboles con ramificaciones más profundas:

```
lobstr::ast(f1(f2(a, b), f3(1, f4(2))))
#>  f1
#>   f2
#>    a
#>    b
#>   f3
#>    1
#>     f4
#>      2
```

Debido a que todas las formas de función se pueden escribir en forma de prefijo (Section 6.8.2), cada expresión R se puede mostrar de esta manera:

```
lobstr::ast(1 + 2 * 3)
#>  `+`
#>  1
#>   `*`
#>    2
#>    3
```

Mostrar el AST de esta manera es una herramienta útil para explorar la gramática de R, el tema de la Section 18.4.

17.4. El código puede generar código

Además de ver el árbol a partir del código escrito por un humano, también puede usar el código para crear nuevos árboles. Hay dos herramientas principales: `call2()` y eliminación de comillas.

`rlang::call2()` construye una llamada de función a partir de sus componentes: la función a llamar y los argumentos para llamarla.

```
call2("f", 1, 2, 3)
#> f(1, 2, 3)
call2("+", 1, call2("*", 2, 3))
#> 1 + 2 * 3
```

`call2()` a menudo es conveniente para programar, pero es un poco torpe para el uso interactivo. Una técnica alternativa es construir árboles de código complejos combinando árboles de código más simples con una plantilla. `expr()` y `enexpr()` tienen soporte incorporado para esta idea a través de `!!` (pronunciado bang-bang), el **operador sin comillas**.

Los detalles precisos son el tema de la Sección @ref(unquoting), pero básicamente `!!x` inserta el árbol de código almacenado en `x` en la expresión. Esto facilita la construcción de árboles complejos a partir de fragmentos simples:

```
xx <- expr(x + x)
yy <- expr(y + y)

expr(!!xx / !!yy)
#> (x + x)/(y + y)
```

Tenga en cuenta que la salida conserva la precedencia del operador, por lo que obtenemos `(x + x) / (y + y)` y no `x + x / y + y` (es decir, `x + (x / y) + y`). Esto es importante, especialmente si te has estado preguntando si no sería más fácil simplemente pegar cadenas.

Quitar las comillas se vuelve aún más útil cuando lo envuelves en una función, primero usando `enexpr()` para capturar la expresión del usuario, luego `expr()` y `!!` para crear una nueva expresión usando una plantilla. El siguiente ejemplo muestra cómo puede generar una expresión que calcule el coeficiente de variación:

```
cv <- function(var) {
  var <- enexpr(var)
  expr(sd(!!var) / mean(!!var))
}

cv(x)
#> sd(x)/mean(x)
cv(x + y)
#> sd(x + y)/mean(x + y)
```

(Esto no es muy útil aquí, pero poder crear este tipo de bloque de construcción es muy útil cuando se resuelven problemas más complejos.)

Es importante destacar que esto funciona incluso cuando se le dan nombres de variables extraños:

```
cv(`)`)
#> sd(`)`)/mean(`)`)
```

Tratar con nombres raros[1] es otra buena razón para evitar `paste()` al generar código R. Puede pensar que se trata de una preocupación esotérica, pero no preocuparse por ello cuando la generación de código SQL en aplicaciones web condujo a ataques de inyección de SQL que, en conjunto, han costado miles de millones de dólares.

17.5. Código de ejecución de evaluación

Inspeccionar y modificar el código le brinda un conjunto de herramientas poderosas. Obtiene otro conjunto de herramientas poderosas cuando **evalúa**, es decir, ejecuta o ejecuta, una expresión. Evaluar una expresión requiere un entorno, que le dice a R qué significan los símbolos en la expresión. Aprenderá los detalles de la evaluación en el Chapter 20.

La herramienta principal para evaluar expresiones es `base::eval()`, que toma una expresión y un entorno:

```
eval(expr(x + y), env(x = 1, y = 10))
#> [1] 11
eval(expr(x + y), env(x = 2, y = 100))
#> [1] 102
```

Si omite el entorno, `eval` usa el entorno actual:

[1] Más técnicamente, estos se denominan nombres no sintácticos y son el tema de la Section 2.2.1.

```
x <- 10
y <- 100
eval(expr(x + y))
#> [1] 110
```

Una de las grandes ventajas de evaluar el código manualmente es que puede modificar el entorno. Hay dos razones principales para hacer esto:

- Para anular temporalmente las funciones para implementar un lenguaje específico de dominio.
- Para agregar una máscara de datos para que pueda hacer referencia a las variables en un data frame como si fueran variables en un entorno.

17.6. Personalización de la evaluación con funciones

El ejemplo anterior usó un entorno que vinculaba `x` e `y` a vectores. Es menos obvio que también vincula nombres a funciones, lo que le permite anular el comportamiento de las funciones existentes. Esta es una gran idea a la que volveremos en el Chapter 21 donde exploro la generación de HTML y LaTeX desde R. El siguiente ejemplo le da una idea del poder. Aquí evalúo el código en un entorno especial donde `*` y `+` han sido anulados para trabajar con cadenas en lugar de números:

```
string_math <- function(x) {
  e <- env(
    caller_env(),
    `+` = function(x, y) paste0(x, y),
    `*` = function(x, y) strrep(x, y)
  )

  eval(enexpr(x), e)
```

```
}

name <- "Hadley"
string_math("Hello " + name)
#> [1] "Hello Hadley"
string_math(("x" * 2 + "-y") * 3)
#> [1] "xx-yxx-yxx-y"
```

dplyr lleva esta idea al extremo, ejecutando código en un entorno que genera SQL para su ejecución en una base de datos remota:

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

con <- DBI::dbConnect(RSQLite::SQLite(), filename = ":memory:")
mtcars_db <- copy_to(con, mtcars)

mtcars_db %>%
  filter(cyl > 2) %>%
  select(mpg:hp) %>%
  head(10) %>%
  show_query()
#> <SQL>
#> SELECT `mpg`, `cyl`, `disp`, `hp`
#> FROM `mtcars`
#> WHERE (`cyl` > 2.0)
```

```
#> LIMIT 10

DBI::dbDisconnect(con)
```

17.7. Personalización de la evaluación con datos

Reenlazar funciones es una técnica extremadamente poderosa, pero tiende a requerir una gran inversión. Una aplicación práctica más inmediata es modificar la evaluación para buscar variables en un data frame en lugar de un entorno. Esta idea impulsa las funciones base `subset()` y `transform()`, así como muchas funciones tidyverse como `ggplot2::aes()` y `dplyr::mutate()`. Es posible usar `eval()` para esto, pero hay algunas trampas potenciales (Section 20.6), así que cambiaremos a `rlang::eval_tidy()` en su lugar.

Además de la expresión y el entorno, `eval_tidy()` también toma una **máscara de datos**, que suele ser un data frame:

```
df <- data.frame(x = 1:5, y = sample(5))
eval_tidy(expr(x + y), df)
#> [1] 6 6 4 6 8
```

Evaluar con una máscara de datos es una técnica útil para el análisis interactivo porque le permite escribir `x + y` en lugar de `df$x + df$y`. Sin embargo, esa conveniencia tiene un costo: la ambigüedad. En la Section 20.4 aprenderá cómo lidiar con la ambigüedad usando los pronombres especiales `.data` y `.env`.

Podemos envolver este patrón en una función usando `enexpr()`. Esto nos da una función muy similar a `base::with()`:

```
with2 <- function(df, expr) {
  eval_tidy(enexpr(expr), df)
}

with2(df, x + y)
#> [1] 6 6 4 6 8
```

Desafortunadamente, esta función tiene un error sutil y necesitamos una nueva estructura de datos para ayudar a solucionarlo.

17.8. Quosures

Para hacer el problema más obvio, voy a modificar `with2()`. El problema básico aún ocurre sin esta modificación, pero es mucho más difícil de ver.

```
with2 <- function(df, expr) {
  a <- 1000
  eval_tidy(enexpr(expr), df)
}
```

Podemos ver el problema cuando usamos `with2()` para referirnos a una variable llamada `a`. Queremos que el valor de `a` provenga del enlace que podemos ver (10), no del enlace interno de la función (1000):

```
df <- data.frame(x = 1:3)
a <- 10
with2(df, x + a)
#> [1] 1001 1002 1003
```

El problema surge porque necesitamos evaluar la expresión capturada en el entorno donde fue escrita (donde `a` es 10), no el entorno dentro de `with2()` (donde `a` es 1000).

Afortunadamente podemos resolver este problema usando una nueva estructura de datos: el **quosure** que agrupa una expresión con un entorno. `eval_tidy()` sabe cómo trabajar con quosures, así que todo lo que tenemos que hacer es cambiar `enexpr()` por `enquo()`:

```
with2 <- function(df, expr) {
  a <- 1000
  eval_tidy(enquo(expr), df)
}

with2(df, x + a)
#> [1] 11 12 13
```

Siempre que utilice una máscara de datos, siempre debe utilizar `enquo()` en lugar de `enexpr()`. Este es el tema del Chapter 20.

18. Expresiones

18.1. Introducción

Para calcular el lenguaje, primero necesitamos entender su estructura. Eso requiere un vocabulario nuevo, algunas herramientas nuevas y algunas formas nuevas de pensar sobre el código R. El primero de ellos es la distinción entre una operación y su resultado. Toma el siguiente código, que multiplica una variable `x` por 10 y guarda el resultado en una nueva variable llamada `y`. No funciona porque no hemos definido una variable llamada `x`:

```
y <- x * 10
#> Error in eval(expr, envir, enclos): object 'x' not found
```

Sería bueno si pudiéramos capturar la intención del código sin ejecutarlo. En otras palabras, ¿cómo podemos separar nuestra descripción de la acción de la acción misma?

Una forma es usar `rlang::expr()`:

```
z <- rlang::expr(y <- x * 10)
z
#> y <- x * 10
```

`expr()` devuelve una expresión, un objeto que captura la estructura del código sin evaluarlo (es decir, ejecutarlo). Si tiene una expresión, puede evaluarla con `base::eval()`:

```
x <- 4
eval(z)
y
#> [1] 40
```

El enfoque de este capítulo son las estructuras de datos que subyacen a las expresiones. Dominar este conocimiento le permitirá inspeccionar y modificar el código capturado y generar código con código. Volveremos a `expr()` en el Chapter 19, ya `eval()` en el Chapter 20.

Estructura

- La Section 18.2 introduce la idea del árbol de sintaxis abstracta (AST) y revela la estructura de árbol que subyace en todo el código R.

- La Section 18.3 se sumerge en los detalles de las estructuras de datos que sustentan el AST: constantes, símbolos y llamadas, que se conocen colectivamente como expresiones.

- La Section 18.4 cubre el análisis, el acto de convertir la secuencia lineal de caracteres en código en AST, y usa esa idea para explorar algunos detalles de la gramática de R.

- La Section 18.5 le muestra cómo puede usar funciones recursivas para calcular en el lenguaje, escribiendo funciones que calculan con expresiones.

- La Section 18.6 vuelve a tres estructuras de datos más especializadas: listas de pares, argumentos perdidos y vectores de expresión.

Requisitos previos

Asegúrese de haber leído la descripción general de la metaprogramación en el Chapter 17 para obtener una descripción general amplia de la motivación y el vocabulario básico. También necesitará el paquete rlang para capturar y calcular expresiones, y el paquete lobstr para visualizarlos.

```
library(rlang)
library(lobstr)
```

18.2. Árboles de sintaxis abstracta

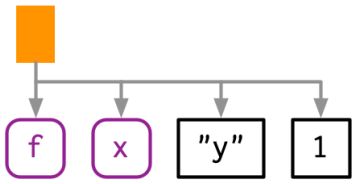
Las expresiones también se denominan **árboles de sintaxis abstracta** (AST) porque la estructura del código es jerárquica y se puede representar naturalmente como un árbol. Comprender esta estructura de árbol es crucial para inspeccionar y modificar expresiones (es decir, metaprogramación).

18.2.1. Dibujo

Comenzaremos presentando algunas convenciones para dibujar AST, comenzando con una simple llamada que muestra sus componentes principales: `f(x, "y", 1)`. Dibujaré árboles de dos maneras[1]:

- A "mano" (es decir, con OmniGraffle):

[1] Para un código más complejo, también puede usar el visor de árboles de RStudio, que no obedece a las mismas convenciones gráficas, pero le permite explorar de forma interactiva grandes AST. Pruébelo con `View(expr(f(x, "y", 1)))`.



- Con `lobstr::ast()`:

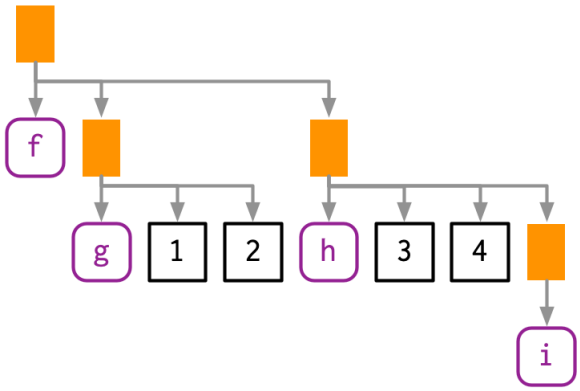
```
lobstr::ast(f(x, "y", 1))
#>  f
#>  x
#>  "y"
#>  1
```

Ambos enfoques comparten convenciones tanto como sea posible:

- Las hojas del árbol son símbolos, como `f` y `x`, o constantes, como `1` o `"y"`. Los símbolos se dibujan en púrpura y tienen esquinas redondeadas. Las constantes tienen bordes negros y esquinas cuadradas. Las cadenas y los símbolos se confunden fácilmente, por lo que las cadenas siempre se escriben entre comillas.
- Las ramas del árbol son objetos de llamada, que representan llamadas de función y se dibujan como rectángulos naranjas. El primer hijo (`f`) es la función que se llama; el segundo hijo y los subsiguientes (`x`, `"y"` y `1`) son los argumentos de esa función.

Los colores se mostrarán cuando *usted* llame a `ast()`, pero no aparecen en el libro por razones técnicas complicadas.

El ejemplo anterior solo contenía una llamada de función, lo que lo convierte en un árbol muy poco profundo. La mayoría de las expresiones contendrán considerablemente más llamadas, creando árboles con múltiples niveles. Por ejemplo, considere el AST para `f(g(1, 2), h(3, 4, i()))`:




```
lobstr::ast(f(g(1, 2), h(3, 4, i())))
#>  f
#>   g
#>    1
#>    2
#>   h
#>    3
#>    4
#>     i
```

Puede leer los diagramas dibujados a mano de izquierda a derecha (ignorando la posición vertical) y los diagramas dibujados por langosta de arriba a abajo (ignorando la posición horizontal). La profundidad dentro del árbol está determinada por el anidamiento de las llamadas a funciones. Esto también determina el orden de evaluación, ya que la evaluación generalmente procede de lo más profundo a lo más superficial, pero esto no está garantizado debido a la evaluación diferida (Section 6.5). También tenga en cuenta la aparición de `i()`, una llamada de función sin argumentos; es una rama con una sola hoja (símbolo).

18.2.2. Componentes sin código

Es posible que se haya preguntado qué hace que estos árboles de sintaxis *abstractos*. Son abstractos porque solo capturan detalles estructurales importantes del código, no espacios en blanco ni comentarios:

```
ast(
  f(x,  y)  # important!
)
#>  f
#>  x
#>  y
```

Solo hay un lugar donde los espacios en blanco afectan el AST:

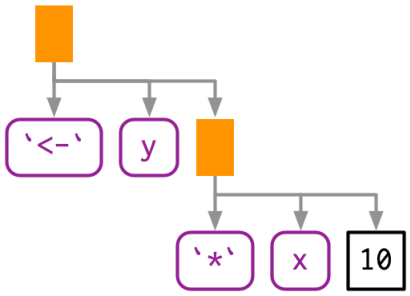
```
lobstr::ast(y <- x)
#>  `<-`
#>  y
#>  x
lobstr::ast(y < -x)
#>  `<`
#>  y
#>   `-`
#>    x
```

18.2.3. Llamadas infijas

Cada llamada en R se puede escribir en forma de árbol porque cualquier llamada se puede escribir en forma de prefijo (Section 6.8.1). Tome `y <- x * 10` de nuevo: ¿cuáles son las funciones que se están llamando? No es tan fácil de detectar como `f(x, 1)` porque esta expresión contiene dos llamadas infijas: `<-` y `*`. Eso significa que estas dos líneas de código son equivalentes:

```
y <- x * 10
`<-`(y, `*`(x, 10))
```

Y ambas tienen este AST[2]:




```
lobstr::ast(y <- x * 10)
#>  `<-`
#>  y
#>   `*`
#>    x
#>    10
```

Realmente no hay diferencia entre los AST, y si genera una expresión con llamadas de prefijo, R aún la imprimirá en forma de infijo:

```
expr(`<-`(y, `*`(x, 10)))
#> y <- x * 10
```

El orden en que se aplican los operadores infijos se rige por un conjunto de reglas llamadas precedencia de operadores, y usaremos `lobstr::ast()` para explorarlas en la Section 18.4.1.

[2]Los nombres de las funciones sin prefijo no son sintácticos, por lo que los rodeo con ``, como en la Section 2.2.1.

18.2.4. Ejercicios

1. Reconstruya el código representado por los árboles a continuación:

```
#>  f
#>   g
#>     h
#>  `+`
#>   `+`
#>    1
#>    2
#>  3
#>  `*`
#>   `(`
#>     `+`
#>      x
#>      y
#>  z
```

2. Dibuja los siguientes árboles a mano y luego verifica tus respuestas con `lobstr::ast()`.

```
f(g(h(i(1, 2, 3))))
f(1, g(2, h(3, i())))
f(g(1, 2), h(3, i(4, 5)))
```

3. ¿Qué está pasando con los AST a continuación? (Sugerencia: lea atentamente `?"^"`.)

```
lobstr::ast(`x` + `y`)
#>  `+`
#>  x
#>  y
lobstr::ast(x ** y)
#>  `^`
```

```
#>  x
#>  y
lobstr::ast(1 -> x)
#>  `<-`
#>  x
#>  1
```

4. ¿Qué está pasando con los AST a continuación? (Sugerencia: lea atentamente la Section 6.2.1.)

```
lobstr::ast(function(x = 1, y = 2) {})
#>  `function`
#>   x = 1
#>    y = 2
#>   `{`
#>  <inline srcref>
```

5. ¿Cómo se ve el árbol de llamadas de una instrucción `if` con múltiples condiciones `else if`? ¿Por qué?

18.3. Expresiones

En conjunto, las estructuras de datos presentes en el AST se denominan expresiones. Una **expresión** es cualquier miembro del conjunto de tipos base creados mediante el código de análisis: escalares constantes, símbolos, objetos de llamada y listas de pares. Estas son las estructuras de datos utilizadas para representar el código capturado de `expr()`, y `is_expression(expr(...))` siempre es verdadero. Las constantes, los símbolos y los objetos de llamada son los más importantes y se analizan a continuación. Las listas de pares y los símbolos vacíos son más especializados y volveremos a ellos en las secciones Section 18.6.1 y Section 18.6.2.

NB: En la documentación base de R, "expresión" se usa para significar dos cosas. Además de la definición anterior, expresión también se usa

para referirse al tipo de objeto devuelto por `expression()` y `parse()`, que son básicamente listas de expresiones como se define anteriormente. En este libro llamaré a estos **vectores de expresión**, y regresaré a ellos en la Section 18.6.3.

18.3.1. Constantes

Las constantes escalares son el componente más simple del AST. Más precisamente, una **constante** es `NULL` o un vector atómico de longitud 1 (o escalar, Section 3.2.1) como `TRUE`, `1L`, `2.5` o `"x"`. Puedes probar una constante con `rlang::is_syntactic_literal()`.

Las constantes son autocomillas en el sentido de que la expresión utilizada para representar una constante es la misma constante:

```
identical(expr(TRUE), TRUE)
#> [1] TRUE
identical(expr(1), 1)
#> [1] TRUE
identical(expr(2L), 2L)
#> [1] TRUE
identical(expr("x"), "x")
#> [1] TRUE
```

18.3.2. Simbolos

Un **símbolo** representa el nombre de un objeto como `x`, `mtcars` o `mean`. En la base R, los términos símbolo y nombre se usan indistintamente (es decir, `is.name()` es idéntico a `is.symbol()`), pero en este libro usé símbolo de manera consistente porque "nombre" tiene muchos otros significados.

Puede crear un símbolo de dos maneras: capturando el código que hace referencia a un objeto con `expr()`, o convirtiendo una cadena en un símbolo con `rlang::sym()`:

```
expr(x)
#> x
sym("x")
#> x
```

Puede volver a convertir un símbolo en una cadena con `as.character()` o `rlang::as_string()`. `as_string()` tiene la ventaja de indicar claramente que obtendrá un vector de caracteres de longitud 1.

```
as_string(expr(x))
#> [1] "x"
```

Puede reconocer un símbolo porque está impreso sin comillas, `str()` le dice que es un símbolo, y `is.symbol()` es `TRUE`:

```
str(expr(x))
#>  symbol x
is.symbol(expr(x))
#> [1] TRUE
```

El tipo de símbolo no está vectorizado, es decir, un símbolo siempre tiene una longitud de 1. Si desea varios símbolos, deberá ponerlos en una lista usando (p. ej.) `rlang::syms()`.

18.3.3. Llamadas

Un **objeto de llamada** representa una llamada de función capturada. Los objetos de llamada son un tipo especial de lista donde el primer componente especifica la función a llamar (generalmente un símbolo), y los

elementos restantes son los argumentos para esa llamada. Los objetos de llamada crean ramas en el AST, porque las llamadas se pueden anidar dentro de otras llamadas.

Puede identificar un objeto de llamada cuando se imprime porque parece una llamada de función. Confusamente `typeof()` y `str()` imprimen "lenguaje" para los objetos de llamada, pero `is.call()` devuelve `TRUE`:

```
lobstr::ast(read.table("important.csv", row.names = FALSE))
#>  read.table
#>  "important.csv"
#>  row.names = FALSE
x <- expr(read.table("important.csv", row.names = FALSE))

typeof(x)
#> [1] "language"
is.call(x)
#> [1] TRUE
```

18.3.3.1. Subconjunto

Las llamadas generalmente se comportan como listas, es decir, puede usar herramientas estándar de creación de subconjuntos. El primer elemento del objeto de llamada es la función a llamar, que suele ser un símbolo:

```
x[[1]]
#> read.table
is.symbol(x[[1]])
#> [1] TRUE
```

El resto de los elementos son los argumentos:

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

Puede extraer argumentos individuales con `[[` o, si se nombra, `$`:

```
x[[2]]
#> [1] "important.csv"
x$row.names
#> [1] FALSE
```

Puede determinar la cantidad de argumentos en un objeto de llamada restando 1 de su longitud:

```
length(x) - 1
#> [1] 2
```

Extraer argumentos específicos de las llamadas es un desafío debido a las reglas flexibles de R para la coincidencia de argumentos: potencialmente podría estar en cualquier ubicación, con el nombre completo, con un nombre abreviado o sin nombre. Para solucionar este problema, puede usar `rlang::call_standardise()` que estandariza todos los argumentos para usar el nombre completo:

```
rlang::call_standardise(x)
#> Warning: `call_standardise()` is deprecated as of rlang 0.4.11
#> This warning is displayed once every 8 hours.
#> read.table(file = "important.csv", row.names = FALSE)
```

(NB: Si la función usa `...` no es posible estandarizar todos los argumentos).

Las llamadas se pueden modificar de la misma forma que las listas:

```
x$header <- TRUE
x
#> read.table("important.csv", row.names = FALSE, header = TRUE)
```

18.3.3.2. Posición de la función

El primer elemento del objeto de llamada es la **posición de la función**. Contiene la función que se llamará cuando se evalúe el objeto, y generalmente es un símbolo [3]:

```
lobstr::ast(foo())
#>  foo
```

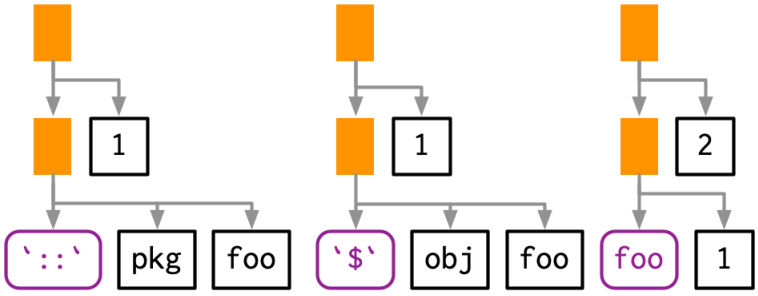
Mientras que R le permite rodear el nombre de la función con comillas, el analizador lo convierte en un símbolo:

```
lobstr::ast("foo"())
#>  foo
```

Sin embargo, a veces la función no existe en el entorno actual y es necesario realizar algunos cálculos para recuperarla: por ejemplo, si la función está en otro paquete, es un método de un objeto R6 o es creada por una fábrica de funciones. En este caso, la posición de la función será ocupada por otra llamada:

[3] Curiosamente, también puede ser un número, como en la expresión `3()`. Pero esta llamada siempre fallará en la evaluación porque un número no es una función.

```
lobstr::ast(pkg::foo(1))
#>    `::`
#>    pkg
#>    foo
#>  1
lobstr::ast(obj$foo(1))
#>    `$`
#>    obj
#>    foo
#>  1
lobstr::ast(foo(1)(2))
#>    foo
#>    1
#>  2
```



18.3.3.3. Construyendo

Puede construir un objeto de llamada a partir de sus componentes utilizando `rlang::call2()`. El primer argumento es el nombre de la función a llamar (ya sea como una cadena, un símbolo u otra llamada). Los argumentos restantes se pasarán a la llamada:

```
call2("mean", x = expr(x), na.rm = TRUE)
#> mean(x = x, na.rm = TRUE)
call2(expr(base::mean), x = expr(x), na.rm = TRUE)
#> base::mean(x = x, na.rm = TRUE)
```

Las llamadas de infijo creadas de esta manera aún se imprimen como de costumbre.

```
call2("<-", expr(x), 10)
#> x <- 10
```

Usar `call2()` para crear expresiones complejas es un poco torpe. Aprenderás otra técnica en el Chapter 19.

18.3.4. Resumen

La siguiente tabla resume la apariencia de los diferentes subtipos de expresión en `str()` y `typeof()`:

	`str()`	`typeof()`
constante escalar	`logi/int/num/chr`	`logical/integer/double/character`
Símbolo	`symbol`	`symbol`
Objeto de llamada	`language`	`language`
Lista de pares	Lista de pares punteados	`pairlist`
Vector de expresión	`expression()`	`expression`

Tanto base R como rlang proporcionan funciones para probar cada tipo de entrada, aunque los tipos cubiertos son ligeramente diferentes. Puede

distinguirlas fácilmente porque todas las funciones básicas comienzan con `is.` y las funciones rlang comienzan con `is_`.

	base	rlang
Scalar constant	—	`is_syntactic_literal()`
Symbol	`is.symbol()`	`is_symbol()`
Call object	`is.call()`	`is_call()`
Pairlist	`is.pairlist()`	`is_pairlist()`
Expression vector	`is.expression()`	—

18.3.5. Ejercicios

1. ¿Cuáles dos de los seis tipos de vectores atómicos no pueden aparecer en una expresión? ¿Por qué? De manera similar, ¿por qué no puedes crear una expresión que contenga un vector atómico de longitud mayor que uno?

2. ¿Qué sucede cuando crea un subconjunto de un objeto de llamada para eliminar el primer elemento? p.ej. `expr(read.csv("foo.csv", header = TRUE))[-1]`. ¿Por qué?

3. Describa las diferencias entre los siguientes objetos de llamada.

```
x <- 1:10

call2(median, x, na.rm = TRUE)
call2(expr(median), x, na.rm = TRUE)
call2(median, expr(x), na.rm = TRUE)
call2(expr(median), expr(x), na.rm = TRUE)
```

4. `rlang::call_standardise()` no funciona tan bien para las siguientes llamadas. ¿Por qué? ¿Qué hace especial a `mean()`?

```
call_standardise(quote(mean(1:10, na.rm = TRUE)))
#> mean(x = 1:10, na.rm = TRUE)
call_standardise(quote(mean(n = T, 1:10)))
#> mean(x = 1:10, n = T)
```

```
call_standardise(quote(mean(x = 1:10, , TRUE)))
#> mean(x = 1:10, , TRUE)
```

5. ¿Por qué este código no tiene sentido?

```
x <- expr(foo(x = 1))
names(x) <- c("x", "y")
```

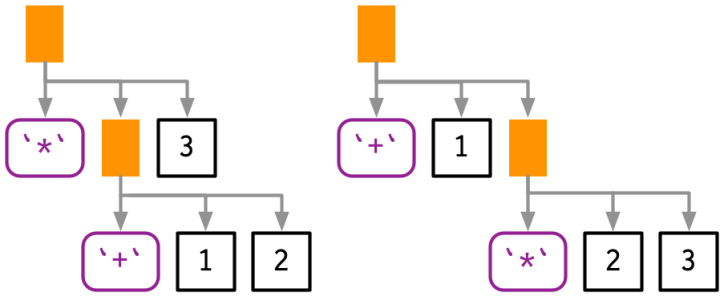
6. Construya la expresión `if(x > 1) "a" else "b"` utilizando varias llamadas a `call2()`. ¿Cómo refleja la estructura del código la estructura del AST?

18.4. Análisis y gramática

Hemos hablado mucho sobre las expresiones y el AST, pero no sobre cómo se crean las expresiones a partir del código que escribe (como `"x + y"`). El proceso mediante el cual un lenguaje informático toma una cadena y construye una expresión se denomina **análisis sintáctico** y se rige por un conjunto de reglas conocido como **gramática**. En esta sección, usaremos `lobstr::ast()` para explorar algunos de los detalles de la gramática de R, y luego mostraremos cómo puede transformar de un lado a otro entre expresiones y cadenas.

18.4.1. Predecencia de operadores

Las funciones infijas introducen dos fuentes de ambigüedad. La primera fuente de ambigüedad surge de las funciones infijas: ¿qué produce `1 + 2 * 3`? ¿Obtienes 9 (es decir, `(1 + 2) * 3`), o 7 (es decir, `1 + (2 * 3)`)? En otras palabras, ¿cuál de los dos posibles árboles de análisis de abajo usa R?



Los lenguajes de programación usan convenciones llamadas **precedencia de operadores** para resolver esta ambigüedad. Podemos usar `ast()` para ver qué hace R:

```
lobstr::ast(1 + 2 * 3)
#>  `+`
#>  1
#>   `*`
#>    2
#>    3
```

Predicting the precedence of arithmetic operations is usually easy because it's drilled into you in school and is consistent across the vast majority of programming languages.

Predecir la precedencia de otros operadores es más difícil. Hay un caso particularmente sorprendente en R: `!` tiene una precedencia mucho menor (es decir, se une con menos fuerza) de lo que cabría esperar. Esto le permite escribir operaciones útiles como:

```
lobstr::ast(!x %in% y)
#>  `!`
#>   `%in%`
```

```
#>    x
#>    y
```

R tiene más de 30 operadores infijos divididos en 18 grupos de precedencia. Si bien los detalles se describen en `?Syntax`, muy pocas personas han memorizado el orden completo. Si hay alguna confusión, ¡use paréntesis!

```
lobstr::ast((1 + 2) * 3)
#>  `*`
#>   `(`
#>    `+`
#>     1
#>     2
#>  3
```

Tenga en cuenta la aparición de los paréntesis en el AST como una llamada a la función `(`.

18.4.2. Asociatividad

La segunda fuente de ambigüedad se presenta por el uso repetido de la misma función de infijo. Por ejemplo, ¿es '1 + 2 + 3' equivalente a '(1 + 2) + 3' o a '1 + (2 + 3)'? Esto normalmente no importa porque `x + (y + z) == (x + y) + z`, es decir, la suma es asociativa, pero es necesaria porque algunas clases de S3 definen `+` de forma no asociativa. Por ejemplo, ggplot2 sobrecarga `+` para construir una trama compleja a partir de piezas simples; esto no es asociativo porque las capas anteriores se dibujan debajo de las capas posteriores (es decir, `geom_point() + geom_smooth()` no produce el mismo gráfico que `geom_smooth() + geom_point()`).

En R, la mayoría de los operadores son **asociativos a la izquierda**, es decir, las operaciones de la izquierda se evalúan primero:

```
lobstr::ast(1 + 2 + 3)
#>  `+`
#>   `+`
#>    1
#>    2
#>  3
```

Hay dos excepciones: exponenciación y asignación.

```
lobstr::ast(2^2^3)
#>  `^`
#>  2
#>   `^`
#>    2
#>    3
lobstr::ast(x <- y <- z)
#>  `<-`
#>  x
#>   `<-`
#>    y
#>    z
```

18.4.3. Analizar y desanalizar

La mayoría de las veces, escribe código en la consola y R se encarga de convertir los caracteres que ha escrito en un AST. Pero ocasionalmente tiene código almacenado en una cadena y desea analizarlo usted mismo. Puedes hacerlo usando `rlang::parse_expr()`:

```
x1 <- "y <- x + 10"
x1
#> [1] "y <- x + 10"
```

```
is.call(x1)
#> [1] FALSE

x2 <- rlang::parse_expr(x1)
x2
#> y <- x + 10
is.call(x2)
#> [1] TRUE
```

`parse_expr()` siempre devuelve una sola expresión. Si tiene varias expresiones separadas por `;` o `\n`, deberá usar `rlang::parse_exprs()`. Devuelve una lista de expresiones:

```
x3 <- "a <- 1; a + 1"
rlang::parse_exprs(x3)
#> [[1]]
#> a <- 1
#>
#> [[2]]
#> a + 1
```

Si se encuentra trabajando con cadenas que contienen código con mucha frecuencia, debe reconsiderar su proceso. Lea el Chapter 19 y considere si puede generar expresiones utilizando la cuasicita de manera más segura.

El equivalente básico de `parse_exprs()` es `parse()`. Es un poco más difícil de usar porque está especializado para analizar código R almacenado en archivos. Debe proporcionar su cadena al argumento `texto` y devolverá un vector de expresión (Section 18.6.3). Recomiendo convertir la salida en una lista:

```
as.list(parse(text = x1))
#> [[1]]
#> y <- x + 10
```

Lo contrario de analizar es **deparsear**: dada una expresión, desea la cadena que la generaría. Esto sucede automáticamente cuando imprime una expresión, y puede obtener la cadena con `rlang::expr_text()`:

```
z <- expr(y <- x + 10)
expr_text(z)
#> [1] "y <- x + 10"
```

El análisis y la eliminación no son perfectamente simétricos porque el análisis genera un árbol de sintaxis *abstracto*. Esto significa que perdemos los acentos graves en los nombres, comentarios y espacios en blanco ordinarios:

```
cat(expr_text(expr({
  # This is a comment
  x <-              `x` + 1
})))
#> {
#>     x <- x + 1
#> }
```

Tenga cuidado al usar el equivalente base R, `deparse()`: devuelve un vector de caracteres con un elemento para cada línea. Siempre que lo use, recuerde que la longitud de la salida puede ser mayor que uno y planifique en consecuencia.

18.4.4. Ejercicios

1. R usa paréntesis de dos maneras ligeramente diferentes, como se ilustra en estas dos llamadas:

```
f((1))
`(`(1 + 1)
```

Compare y contraste los dos usos haciendo referencia al AST.

2. `=` también se puede utilizar de dos maneras. Construya un ejemplo simple que muestre ambos usos.

3. ¿`-2^2` produce 4 o -4? ¿Por qué?

4. ¿Qué devuelve `!1 + !1`? ¿Por qué?

5. ¿Por qué `x1 <- x2 <- x3 <- 0` funciona? Describe las dos razones.

6. Compara los AST de `x + y %+% z` y `x ^ y %+% z`. ¿Qué has aprendido sobre la precedencia de las funciones de infijo personalizadas?

7. ¿Qué sucede si llamas a `parse_expr()` con una cadena que genera múltiples expresiones? p.ej. `parse_expr("x + 1; y + 1")`

8. ¿Qué sucede si intenta analizar una expresión no válida? p.ej. `"a +"` o `"f())"`.

9. `deparse()` produce vectores cuando la entrada es larga. Por ejemplo, la siguiente llamada produce un vector de longitud dos:

```
expr <- expr(g(a + b + c + d + e + f + g + h + i + j + k + l +
  m + n + o + p + q + r + s + t + u + v + w + x + y + z))

deparse(expr)
```

¿Qué hace `expr_text()` en su lugar?

10. `pairwise.t.test()` asume que `deparse()` siempre devuelve un vector de un carácter de longitud. ¿Puedes construir una entrada que viole esta expectativa? ¿Lo que sucede?

18.5. Walking AST con funciones recursivas

Para concluir el capítulo, voy a utilizar todo lo que ha aprendido sobre los AST para resolver problemas más complicados. La inspiración proviene del paquete de herramientas de código base, que proporciona dos funciones interesantes:

- `findGlobals()` localiza todas las variables globales utilizadas por una función. Esto puede ser útil si desea verificar que su función no dependa inadvertidamente de variables definidas en su entorno principal.

- `checkUsage()` comprueba una variedad de problemas comunes, incluidas las variables locales no utilizadas, los parámetros no utilizados y el uso de coincidencias de argumentos parciales.

Obtener todos los detalles de estas funciones correctamente es complicado, por lo que no desarrollaremos completamente las ideas. En su lugar, nos centraremos en la gran idea subyacente: la recursividad en el AST. Las funciones recursivas se ajustan naturalmente a las estructuras de datos de tipo árbol porque una función recursiva se compone de dos partes que corresponden a las dos partes del árbol:

- El **caso recursivo** maneja los nodos en el árbol. Por lo general, hará algo con cada hijo de un nodo, por lo general llamando a la función recursiva nuevamente, y luego combinará los resultados nuevamente. Para las expresiones, deberá manejar llamadas y listas de pares (argumentos de función).

- El **caso base** maneja las hojas del árbol. Los casos base aseguran que la función eventualmente termine, resolviendo directamente los casos más simples. Para las expresiones, debe manejar símbolos y constantes en el caso base.

Para que este patrón sea más fácil de ver, necesitaremos dos funciones auxiliares. Primero definimos `expr_type()` que devolverá "constante" para constante, "símbolo" para símbolos, "call", para llamadas, "pairlist" para listas de pares y el "tipo" de cualquier otra cosa:

```
expr_type <- function(x) {
  if (rlang::is_syntactic_literal(x)) {
    "constant"
  } else if (is.symbol(x)) {
    "symbol"
  } else if (is.call(x)) {
    "call"
  } else if (is.pairlist(x)) {
    "pairlist"
  } else {
    typeof(x)
  }
}

expr_type(expr("a"))
#> [1] "constant"
expr_type(expr(x))
#> [1] "symbol"
expr_type(expr(f(1, 2)))
#> [1] "call"
```

Combinaremos esto con un contenedor alrededor de la función de cambio:

```
switch_expr <- function(x, ...) {
  switch(expr_type(x),
    ...,
    stop("Don't know how to handle type ", typeof(x), call. = FALSE)
```

```
  )
}
```

Con estas dos funciones en la mano, podemos escribir una plantilla básica para cualquier función que recorra el AST usando `switch()` (Section 5.2.3):

```
recurse_call <- function(x) {
  switch_expr(x,
    # Casos base
    symbol = ,
    constant = ,

    # Casos recursivos
    call = ,
    pairlist =
  )
}
```

Por lo general, resolver el caso base es fácil, así que lo haremos primero y luego verificaremos los resultados. Los casos recursivos son más complicados y, a menudo, requerirán alguna programación funcional.

18.5.1. Encontrar F y T

Comenzaremos con una función que determina si otra función usa las abreviaturas lógicas `T` y `F` porque usarlas a menudo se considera una mala práctica de codificación. Nuestro objetivo es devolver `TRUE` si la entrada contiene una abreviatura lógica y `FALSE` en caso contrario.

Primero encontremos el tipo de `T` versus `TRUE`:

```
expr_type(expr(TRUE))
#> [1] "constant"

expr_type(expr(T))
#> [1] "symbol"
```

`TRUE` se analiza como un vector lógico de longitud uno, mientras que `T` se analiza como un nombre. Esto nos dice cómo escribir nuestros casos base para la función recursiva: una constante nunca es una abreviatura lógica, y un símbolo es una abreviatura si es "F" o "T":

```
logical_abbr_rec <- function(x) {
  switch_expr(x,
    constant = FALSE,
    symbol = as_string(x) %in% c("F", "T")
  )
}

logical_abbr_rec(expr(TRUE))
#> [1] FALSE
logical_abbr_rec(expr(T))
#> [1] TRUE
```

He escrito la función `logical_abbr_rec()` asumiendo que la entrada será una expresión, ya que esto simplificará la operación recursiva. Sin embargo, cuando se escribe una función recursiva, es común escribir un contenedor que proporciona valores predeterminados o hace que la función sea un poco más fácil de usar. Aquí normalmente crearemos un envoltorio que cita su entrada (aprenderemos más sobre eso en el próximo capítulo), por lo que no necesitamos usar `expr()` cada vez.

```
logical_abbr <- function(x) {
  logical_abbr_rec(enexpr(x))
}

logical_abbr(T)
#> [1] TRUE
logical_abbr(FALSE)
#> [1] FALSE
```

A continuación, debemos implementar los casos recursivos. Aquí queremos hacer lo mismo para las llamadas y para las listas de pares: aplique recursivamente la función a cada subcomponente y devuelva `TRUE` si algún subcomponente contiene una abreviatura lógica. Esto se facilita con `purrr::some()`, que itera sobre una lista y devuelve `TRUE` si la función de predicado es verdadera para cualquier elemento.

```
logical_abbr_rec <- function(x) {
  switch_expr(x,
    # Casos base
    constant = FALSE,
    symbol = as_string(x) %in% c("F", "T"),

    # Casos recursivos
    call = ,
    pairlist = purrr::some(x, logical_abbr_rec)
  )
}

logical_abbr(mean(x, na.rm = T))
#> [1] TRUE
logical_abbr(function(x, na.rm = T) FALSE)
#> [1] TRUE
```

18.5.2. Encontrar todas las variables creadas por asignación

`logical_abbr()` es relativamente simple: solo devuelve un solo `TRUE` o `FALSE`. La siguiente tarea, enumerar todas las variables creadas por asignación, es un poco más complicada. Comenzaremos de manera simple y luego haremos que la función sea progresivamente más rigurosa.

Comenzamos mirando el AST para la asignación:

```
ast(x <- 10)
#>  `<-`
#>  x
#>  10
```

La asignación es un objeto de llamada donde el primer elemento es el símbolo `<-`, el segundo es el nombre de la variable y el tercero es el valor a asignar.

A continuación, debemos decidir qué estructura de datos vamos a utilizar para los resultados. Aquí creo que será más fácil si devolvemos un vector de caracteres. Si devolvemos símbolos, necesitaremos usar una `list()` y eso hace las cosas un poco más complicadas.

Con eso en la mano, podemos comenzar implementando los casos base y proporcionando un envoltorio útil alrededor de la función recursiva. Aquí los casos base son sencillos porque sabemos que ni un símbolo ni una constante representan una asignación.

```
find_assign_rec <- function(x) {
  switch_expr(x,
    constant = ,
    symbol = character()
  )
}
find_assign <- function(x) find_assign_rec(enexpr(x))
```

```
find_assign("x")
#> character(0)
find_assign(x)
#> character(0)
```

A continuación implementamos los casos recursivos. Esto es más fácil gracias a una función que debería existir en purrr, pero actualmente no existe. `flat_map_chr()` espera que `.f` devuelva un vector de caracteres de longitud arbitraria y aplana todos los resultados en un solo vector de caracteres.

```
flat_map_chr <- function(.x, .f, ...) {
  purrr::flatten_chr(purrr::map(.x, .f, ...))
}

flat_map_chr(letters[1:3], ~ rep(., sample(3, 1)))
#> [1] "a" "b" "b" "b" "c" "c" "c"
```

El caso recursivo para las listas de pares es sencillo: iteramos sobre cada elemento de la lista de pares (es decir, cada argumento de función) y combinamos los resultados. El caso de las llamadas es un poco más complejo: si se trata de una llamada a `<-` entonces deberíamos devolver el segundo elemento de la llamada:

```
find_assign_rec <- function(x) {
  switch_expr(x,
    # Casos base
    constant = ,
    symbol = character(),

    # Casos recursivos
    pairlist = flat_map_chr(as.list(x), find_assign_rec),
```

```
    call = {
      if (is_call(x, "<-")) {
        as_string(x[[2]])
      } else {
        flat_map_chr(as.list(x), find_assign_rec)
      }
    }
  )
}

find_assign(a <- 1)
#> [1] "a"
find_assign({
  a <- 1
  {
    b <- 2
  }
})
#> [1] "a" "b"
```

Ahora necesitamos hacer que nuestra función sea más robusta al presentar ejemplos destinados a romperla. ¿Qué sucede cuando asignamos a la misma variable varias veces?

```
find_assign({
  a <- 1
  a <- 2
})
#> [1] "a" "a"
```

Es más fácil arreglar esto en el nivel de la función contenedora:

```
find_assign <- function(x) unique(find_assign_rec(enexpr(x)))

find_assign({
  a <- 1
  a <- 2
})
#> [1] "a"
```

¿Qué sucede si tenemos llamadas anidadas a `<-`? Actualmente solo devolvemos el primero. Eso es porque cuando ocurre `<-` terminamos inmediatamente la recursividad.

```
find_assign({
  a <- b <- c <- 1
})
#> [1] "a"
```

En su lugar, tenemos que adoptar un enfoque más riguroso. Creo que es mejor mantener la función recursiva enfocada en la estructura de árbol, así que voy a extraer `find_assign_call()` en una función separada.

```
find_assign_call <- function(x) {
  if (is_call(x, "<-") && is_symbol(x[[2]])) {
    lhs <- as_string(x[[2]])
    children <- as.list(x)[-1]
  } else {
    lhs <- character()
    children <- as.list(x)
  }

  c(lhs, flat_map_chr(children, find_assign_rec))
}
```

```
find_assign_rec <- function(x) {
  switch_expr(x,
    # Casos base
    constant = ,
    symbol = character(),

    # Casos recursivos
    pairlist = flat_map_chr(x, find_assign_rec),
    call = find_assign_call(x)
  )
}

find_assign(a <- b <- c <- 1)
#> [1] "a" "b" "c"
find_assign(system.time(x <- print(y <- 5)))
#> [1] "x" "y"
```

La versión completa de esta función es bastante complicada, es importante recordar que la escribimos trabajando a nuestro modo escribiendo componentes simples.

18.5.3. Ejercicios

1. `logical_abbr()` devuelve `TRUE` para `T(1, 2, 3)`. ¿Cómo podrías modificar `logical_abbr_rec()` para que ignore las llamadas a funciones que usan `T` o `F`?

2. `logical_abbr()` trabaja con expresiones. Actualmente falla cuando le das una función. ¿Por qué? ¿Cómo podrías modificar `logical_abbr()` para que funcione? ¿Sobre qué componentes de una función necesitará recurrir?

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

3. Modifique `find_assign` para detectar también la asignación usando funciones de reemplazo, es decir, `names(x) <- y`.

4. Escriba una función que extraiga todas las llamadas a una función específica.

18.6. Estructuras de datos especializadas

Hay dos estructuras de datos y un símbolo especial que debemos cubrir en aras de la exhaustividad. No suelen ser importantes en la práctica.

18.6.1. Listas de pares

Las listas de pares son un remanente del pasado de R y han sido reemplazadas por listas en casi todas partes. El único lugar donde es probable que vea listas de pares en R[4] es cuando trabaja con llamadas a la función `función`, ya que los argumentos formales de una función se almacenan en una lista de pares:

```
f <- expr(function(x, y = 10) x + y)

args <- f[[2]]
args
#> $x
#>
```

[4]Si está trabajando en C, encontrará listas de pares con más frecuencia. Por ejemplo, los objetos de llamada también se implementan mediante listas de pares.

```
#>
#> $y
#> [1] 10
typeof(args)
#> [1] "pairlist"
```

Afortunadamente, cada vez que encuentre una lista de pares, puede tratarla como una lista normal:

```
pl <- pairlist(x = 1, y = 2)
length(pl)
#> [1] 2
pl$x
#> [1] 1
```

Detrás de escena, las listas de pares se implementan utilizando una estructura de datos diferente, una lista vinculada en lugar de una matriz. Eso hace que subdividir una lista de pares sea mucho más lento que subdividir una lista, pero esto tiene poco impacto práctico.

18.6.2. Argumentos faltantes

El símbolo especial que necesita un poco más de discusión es el símbolo vacío, que se usa para representar argumentos faltantes (¡no valores faltantes!). Solo necesita preocuparse por el símbolo faltante si está creando funciones mediante programación con argumentos faltantes; volveremos a eso en la Section 19.4.3.

Puedes crear un símbolo vacío con `missing_arg()` (o `expr()`):

```
missing_arg()
typeof(missing_arg())
#> [1] "symbol"
```

Un símbolo vacío no imprime nada, así que puedes comprobar si tienes uno con `rlang::is_missing()`:

```
is_missing(missing_arg())
#> [1] TRUE
```

Los encontrará en la naturaleza en funciones formales:

```
f <- expr(function(x, y = 10) x + y)
args <- f[[2]]
is_missing(args[[1]])
#> [1] TRUE
```

Esto es particularmente importante para `...` que siempre está asociado con un símbolo vacío:

```
f <- expr(function(...) list(...))
args <- f[[2]]
is_missing(args[[1]])
#> [1] TRUE
```

El símbolo vacío tiene una propiedad peculiar: si lo vincula a una variable, luego accede a esa variable, obtendrá un error:

```
m <- missing_arg()
m
#> Error in eval(expr, envir, enclos): argument "m" is missing, with no defau
```

¡Pero no lo hará si lo almacena dentro de otra estructura de datos!

```
ms <- list(missing_arg(), missing_arg())
ms[[1]]
```

Si necesita preservar la falta de una variable, `rlang::maybe_missing()` suele ser útil. Le permite referirse a una variable potencialmente faltante sin desencadenar el error. Consulte la documentación para casos de uso y más detalles.

18.6.3. Vectores de expresión

Finalmente, necesitamos discutir brevemente el vector de expresión. Los vectores de expresión solo son producidos por dos funciones base: `expression()` y `parse()`:

```
exp1 <- parse(text = c("
x <- 4
x
"))
exp2 <- expression(x <- 4, x)

typeof(exp1)
#> [1] "expression"
typeof(exp2)
#> [1] "expression"

exp1
#> expression(x <- 4, x)
exp2
#> expression(x <- 4, x)
```

Al igual que las llamadas y las listas de pares, los vectores de expresión se comportan como listas:

```
length(exp1)
#> [1] 2
exp1[[1]]
#> x <- 4
```

Conceptualmente, un vector de expresión es solo una lista de expresiones. La única diferencia es que llamar a `eval()` en una expresión evalúa cada expresión individual. No creo que esta ventaja merezca la introducción de una nueva estructura de datos, por lo que en lugar de vectores de expresión, solo uso listas de expresiones.

19. Cuasicita

19.1. Introducción

Ahora que comprende la estructura de árbol del código R, es hora de volver a una de las ideas fundamentales que hacen que 'expr()' y 'ast()' funcionen: citar (poner entre comillas). En una evaluación ordenada, todas las funciones de comillas son en realidad funciones de cuasicita porque también admiten la eliminación de comillas. Donde citar es el acto de capturar una expresión no evaluada, **no citar** es la capacidad de evaluar selectivamente partes de una expresión entrecomillada. En conjunto, esto se llama cuasicitar. La cuasicita facilita la creación de funciones que combinan código escrito por el autor de la función con código escrito por el usuario de la función. Esto ayuda a resolver una amplia variedad de problemas desafiantes.

La cuasicita es uno de los tres pilares de la evaluación ordenada. Aprenderá sobre los otros dos (cuotas y máscara de datos) en el Chapter 20. Cuando se usa sola, la cuasicita es más útil para la programación, particularmente para generar código. Pero cuando se combina con otras técnicas, la evaluación ordenada se convierte en una poderosa herramienta para el análisis de datos.

Estructura

- La Section 19.2 motiva el desarrollo de la cuasicita con una función, `cement()`, que funciona como `paste()` pero cita automáticamente

sus argumentos para que usted no tenga que hacerlo.

- La Section 19.3 le brinda las herramientas para citar expresiones, ya sea que provengan de usted o del usuario, o si usa herramientas rlang o base R.
- La Section 19.4 introduce la mayor diferencia entre las funciones de citar de rlang y la función de citar base: quitar las comillas con `!!` y `!!!`.
- La Section 19.5 analiza las tres técnicas principales sin comillas que utilizan las funciones base de R para deshabilitar el comportamiento de comillas.
- La Section 19.6 explora otro lugar donde puedes usar `!!!`, funciones que toman `...`. También presenta el operador especial `:=`, que le permite cambiar dinámicamente los nombres de los argumentos.
- La Section 19.7 algunos usos prácticos de las comillas para resolver problemas que naturalmente requieren cierta generación de código.
- La Section 19.8 termina con un poco de historia de la cuasicitación para aquellos que estén interesados.

Requisitos previos

Asegúrese de haber leído la descripción general de metaprogramación en el Chapter 17 para obtener una descripción general amplia de la motivación y el vocabulario básico, y que esté familiarizado con la estructura de árbol de las expresiones como se describe en la Section 18.3.

En cuanto al código, usaremos principalmente las herramientas de rlang, pero al final del capítulo también verá algunas aplicaciones poderosas junto con purrr.

```
library(rlang)
library(purrr)
```

Trabajo relacionado

Las funciones de citar tienen profundas conexiones con las **macros** de Lisp. Pero las macros generalmente se ejecutan en tiempo de compilación, que no existe en R, y siempre ingresan y generan AST. Consulte Lumley (2001) para conocer un enfoque para implementarlas en R. Las funciones de citar están más estrechamente relacionadas con las funciones más esotéricas de Lisp **fexprs**, donde todos los argumentos se citan por defecto. Es útil conocer estos términos cuando se busca trabajo relacionado en otros lenguajes de programación.

19.2. Motivación

Comenzaremos con un ejemplo concreto que ayuda a motivar la necesidad de eliminar las comillas y, por lo tanto, de las cuasicitas. Imagina que estás creando muchas cadenas uniendo palabras:

```
paste("Good", "morning", "Hadley")
#> [1] "Good morning Hadley"
paste("Good", "afternoon", "Alice")
#> [1] "Good afternoon Alice"
```

Estás harto y cansado de escribir todas esas comillas y, en cambio, solo quieres usar palabras simples. Con ese fin, ha escrito la siguiente función. (No se preocupe por la implementación por ahora; aprenderá sobre las piezas más adelante).

```
cement <- function(...) {
  args <- ensyms(...)
  paste(purrr::map(args, as_string), collapse = " ")
}

cement(Good, morning, Hadley)
#> [1] "Good morning Hadley"
cement(Good, afternoon, Alice)
#> [1] "Good afternoon Alice"
```

Formalmente, esta función cita todas sus entradas. Puede pensar en ello como poner automáticamente comillas alrededor de cada argumento. Eso no es exactamente cierto ya que los objetos intermedios que genera son expresiones, no cadenas, pero es una aproximación útil y el significado raíz del término "cita".

Esta función es buena porque ya no necesitamos escribir comillas. El problema viene cuando queremos usar variables. Es fácil usar variables con `paste()`: simplemente no las rodee con comillas.

```
name <- "Hadley"
time <- "morning"

paste("Good", time, name)
#> [1] "Good morning Hadley"
```

Obviamente, esto no funciona con `cement()` porque cada entrada se cita automáticamente:

```
cement(Good, time, name)
#> [1] "Good time name"
```

Necesitamos alguna forma de *dejar de citar* explícitamente la entrada para decirle a `cement()` que elimine las comillas automáticas. Aquí necesitamos que `time` y `name` se traten de manera diferente a `Good`. Cuasicitación nos da una herramienta estándar para hacerlo: `!!`, llamado "sin comillas", y pronunciado bang-bang. `!!` le dice a una función de comillas que elimine las comillas implícitas:

```
cement(Good, !!time, !!name)
#> [1] "Good morning Hadley"
```

Es útil comparar `cement()` y `paste()` directamente. `paste()` evalúa sus argumentos, por lo que debemos citar donde sea necesario; `cement()` cita sus argumentos, por lo que debemos eliminar las comillas donde sea necesario.

```
paste("Good", time, name)
cement(Good, !!time, !!name)
```

19.2.1. Vocabulario

La distinción entre argumentos citados y evaluados es importante:

- Un argumento **evaluado** obedece las reglas de evaluación usuales de R.
- Un argumento **citado** es capturado por la función y lo procesa de manera personalizada.

`paste()` evalúa todos sus argumentos; `cement()` cita todos sus argumentos.

Si alguna vez no está seguro de si un argumento se cita o se evalúa, intente ejecutar el código fuera de la función. Si no funciona o hace algo diferente,

entonces se cita ese argumento. Por ejemplo, puede usar esta técnica para determinar que se cita el primer argumento de `library()`:

```
# Funciona
library(MASS)

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

Hablar sobre si un argumento se cita o se evalúa es una forma más precisa de establecer si una función utiliza o no una evaluación no estándar (NSE). A veces usaré "función de comillas" como abreviatura de una función que cita uno o más argumentos, pero en general, hablaré de argumentos citados, ya que ese es el nivel en el que se aplica la diferencia.

19.2.2. Ejercicios

1. Para cada función en el siguiente código base de R, identifique qué argumentos se citan y cuáles se evalúan.

   ```
   library(MASS)

   mtcars2 <- subset(mtcars, cyl == 4)

   with(mtcars2, sum(vs))
   sum(mtcars2$am)

   rm(mtcars2)
   ```

2. Para cada función en el siguiente código tidyverse, identifique qué argumentos se citan y cuáles se evalúan.

```
library(dplyr)
library(ggplot2)

by_cyl <- mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(mpg))

ggplot(by_cyl, aes(cyl, mean)) + geom_point()
```

19.3. Citar

La primera parte de la cuasicita es la cita: capturar una expresión sin evaluarla. Necesitaremos un par de funciones porque la expresión se puede proporcionar directa o indirectamente, a través de un argumento de función evaluado de forma perezosa. Comenzaré con las funciones de citar de rlang, luego regresaré a las proporcionadas por la base R.

19.3.1. Captura de expresiones

Hay cuatro funciones de citar importantes. Para la exploración interactiva, el más importante es `expr()`, que captura su argumento exactamente como se proporciona:

```
expr(x + y)
#> x + y
expr(1 / 2 / 3)
#> 1/2/3
```

(Recuerde que los espacios en blanco y los comentarios no forman parte de la expresión, por lo que no serán capturados por una función de comillas.)

`expr()` es excelente para la exploración interactiva, porque captura lo que usted, el desarrollador, escribió. No es tan útil dentro de una función:

```
f1 <- function(x) expr(x)
f1(a + b + c)
#> x
```

Necesitamos otra función para resolver este problema: `enexpr()`. Esto captura lo que la persona que llama proporcionó a la función mirando el objeto de promesa interno que impulsa la evaluación perezosa. (Section 6.5.1).

```
f2 <- function(x) enexpr(x)
f2(a + b + c)
#> a + b + c
```

(Se llama "en"-`expr()` por analogía para enriquecer. Enriquecer a alguien lo hace más rico; `enexpr()` un argumento lo convierte en una expresión.)

Para capturar todos los argumentos en `...`, use `enexprs()`.

```
f <- function(...) enexprs(...)
f(x = 1, y = 10 * z)
#> $x
#> [1] 1
#>
#> $y
#> 10 * z
```

Finalmente, `exprs()` es útil de forma interactiva para hacer una lista de expresiones:

```
exprs(x = x ^ 2, y = y ^ 3, z = z ^ 4)
# Forma abreviada de
# list(x = expr(x ^ 2), y = expr(y ^ 3), z = expr(z ^ 4))
```

En resumen, use `enexpr()` y `enexprs()` para capturar las expresiones suministradas como argumentos *por el usuario*. Usa `expr()` y `exprs()` para capturar expresiones que *tú* proporcionas.

19.3.2. Captura de símbolos

A veces, solo desea permitir que el usuario especifique un nombre de variable, no una expresión arbitraria. En este caso, puede usar `ensym()` o `ensyms()`. Estas son variantes de `enexpr()` y `enexprs()` que comprueban que la expresión capturada es un símbolo o una cadena (que se convierte en un símbolo[1]). `ensym()` y `ensyms()` arrojan un error si se les da algo más.

```
f <- function(...) ensyms(...)
f(x)
#> [[1]]
#> x
f("x")
#> [[1]]
#> x
```

19.3.3. Con R base

Cada función rlang descrita anteriormente tiene un equivalente en base R. Su principal diferencia es que los equivalentes base no admiten la elimi-

[1] Esto es por compatibilidad con la base R, que le permite proporcionar una cadena en lugar de un símbolo en muchos lugares: `"x" <- 1`, `"foo"(x, y)`, `c("x" = 1)`.

nación de comillas (de lo que hablaremos muy pronto). Esto las convierte en funciones de citar, en lugar de funciones de cuasicitación.

El equivalente básico de `expr()` es `quote()`:

```
quote(x + y)
#> x + y
```

La función base más cercana a `enexpr()` es `substitute()`:

```
f3 <- function(x) substitute(x)
f3(x + y)
#> x + y
```

La base equivalente a `exprs()` es `alist()`:

```
alist(x = 1, y = x + 2)
#> $x
#> [1] 1
#>
#> $y
#> x + 2
```

El equivalente a `enexprs()` es una característica no documentada de `substitute()`[2]:

```
f <- function(...) as.list(substitute(...()))
f(x = 1, y = 10 * z)
#> $x
#> [1] 1
#>
#> $y
#> 10 * z
```

[2]Descubierto por Peter Meilstrup y descrito en R-devel el 2018-08-13.

Hay otras dos importantes funciones de citar base que cubriremos en otra parte:

- `bquote()` proporciona una forma limitada de cuasicitación, y se analiza en la Section 19.5.

- `~`, la fórmula, es una función de citar que también captura el entorno. Es la inspiración para quosures, el tema del próximo capítulo, y se discute en Section 20.3.4.

19.3.4. Sustitución

La mayoría de las veces verá que se usa `substitute()` para capturar argumentos no evaluados. Sin embargo, además de citar, `substitute()` también hace sustitución (¡como sugiere su nombre!). Si le da una expresión, en lugar de un símbolo, sustituirá los valores de los símbolos definidos en el entorno actual.

```
f4 <- function(x) substitute(x * 2)
f4(a + b + c)
#> (a + b + c) * 2
```

Creo que esto hace que el código sea difícil de entender, porque si se saca de contexto, no se puede saber si el objetivo de `substitute(x + y)` es reemplazar `x`, `y` o ambos. Si desea usar `substitute()` para la sustitución, le recomiendo que use el segundo argumento para dejar claro su objetivo:

```
substitute(x * y * z, list(x = 10, y = quote(a + b)))
#> 10 * (a + b) * z
```

19.3.5. Resumen

Al citar (es decir, capturar código), hay dos distinciones importantes:

- ¿Lo proporciona el desarrollador del código o el usuario del código? En otras palabras, ¿es fijo (suministrado en el cuerpo de la función) o variable (suministrado a través de un argumento)?
- ¿Quieres capturar una sola expresión o múltiples expresiones?

Esto conduce a una tabla de funciones de 2 × 2 para rlang, Table 19.1, y para R base, Table 19.2.

Table 19.1.: Funciones de cuasicitar de rlang

	Desarrollador	Usario
Uno	`expr()`	`enexpr()`
Muchos	`exprs()`	`enexprs()`

Table 19.2.: Fuciones de cuasicitar de R base

	Desarrollador	Usario
Uno	`quote()`	`substitute()`
Muchos	`alist()`	`as.list(substitute(...()))`

19.3.6. Ejercicios

1. ¿Cómo se implementa `expr()`? Mira su código fuente.
2. Compara y contrasta las siguientes dos funciones. ¿Puedes predecir la salida antes de ejecutarlos?

```
f1 <- function(x, y) {
  exprs(x = x, y = y)
}
f2 <- function(x, y) {
  enexprs(x = x, y = y)
}
f1(a + b, c + d)
f2(a + b, c + d)
```

3. ¿Qué sucede si intenta usar `enexpr()` con una expresión (es decir, `enexpr(x + y)`? ¿Qué sucede si `enexpr()` recibe un argumento faltante?

4. ¿En qué se diferencian `exprs(a)` y `exprs(a = )`? Piensa tanto en la entrada como en la salida.

5. ¿Cuáles son otras diferencias entre `exprs()` y `alist()`? Lea la documentación de los argumentos con nombre de `exprs()` para averiguarlo.

6. La documentación para `substitute()` dice:

   > La sustitución se lleva a cabo examinando cada componente del árbol de análisis de la siguiente manera:
   >
   > - Si no es un símbolo enlazado en `env`, no cambia.
   > - Si es un objeto de promesa (es decir, un argumento formal para una función), el espacio de expresión de la promesa reemplaza al símbolo.
   > - Si es una variable ordinaria, se sustituye su valor, a menos que `env` sea .GlobalEnv, en cuyo caso el símbolo no se modifica.

   Cree ejemplos que ilustren cada uno de los casos anteriores.

19.4. Remover cita

Hasta ahora, solo ha visto ventajas relativamente pequeñas de las funciones de citar de rlang sobre las funciones de citar de base R: tienen un esquema de nomenclatura más consistente. La gran diferencia es que las funciones de comillas de rlang son en realidad funciones de cuasicita porque también pueden quitar las comillas.

Quitar las comillas le permite evaluar de forma selectiva partes de la expresión que, de lo contrario, se citarían, lo que le permite fusionar AST con una plantilla AST. Dado que las funciones base no usan la eliminación de comillas, en su lugar usan una variedad de otras técnicas, que aprenderá en la Section 19.5.

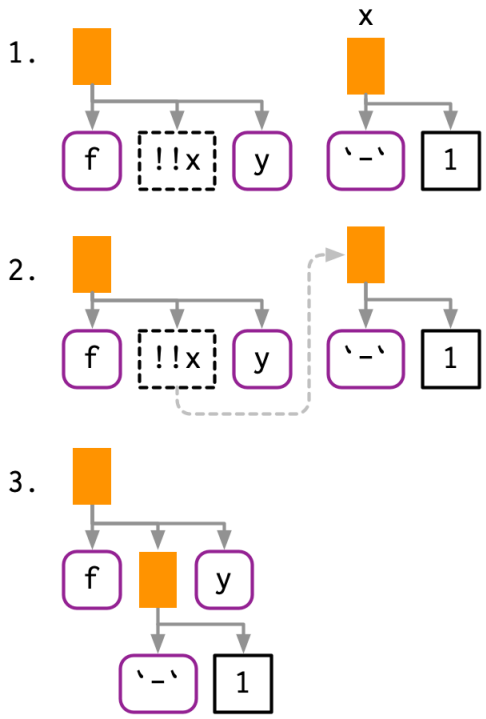
Quitar las comillas te permite evaluar selectivamente el código dentro de `expr()`, de modo que `expr(!!x)` sea equivalente a `x`. En el Chapter 20, aprenderá sobre otra evaluación inversa. Esto sucede fuera de `expr()`, por lo que `eval(expr(x))` es equivalente a `x`.

19.4.1. Remover cita de un argumento

Use `!!` para eliminar las comillas de un solo argumento en una llamada de función. `!!` toma una sola expresión, la evalúa y alinea el resultado en el AST.

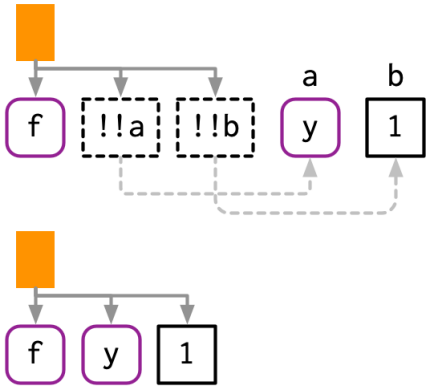
```
x <- expr(-1)
expr(f(!!x, y))
#> f(-1, y)
```

Creo que esto es más fácil de entender con un diagrama. `!!` introduce un marcador de posición en el AST, que se muestra con bordes punteados. Aquí, el marcador de posición `x` se reemplaza por un AST, ilustrado por una conexión punteada.



Además de llamar a objetos, `!!` también funciona con símbolos y constantes:

```
a <- sym("y")
b <- 1
expr(f(!!a, !!b))
#> f(y, 1)
```

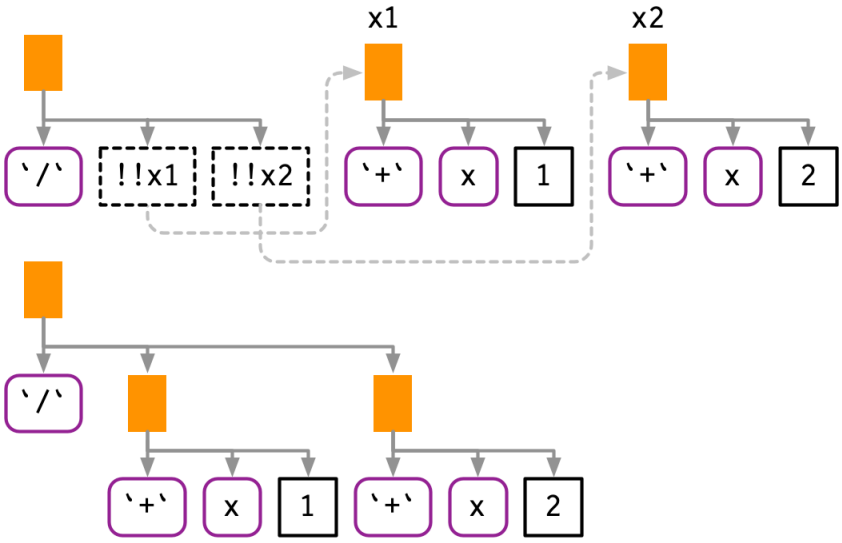


Si el lado derecho de `!!` es una llamada de función, `!!` lo evaluará e insertará los resultados:

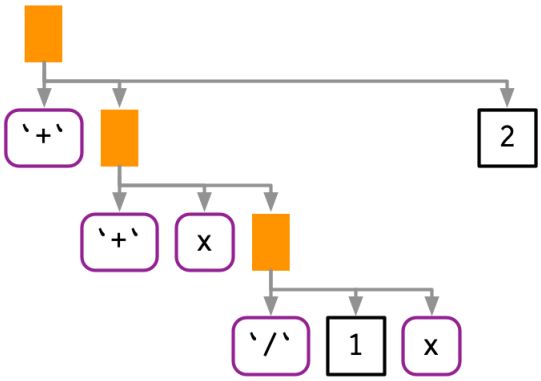
```
mean_rm <- function(var) {
  var <- ensym(var)
  expr(mean(!!var, na.rm = TRUE))
}
expr(!!mean_rm(x) + !!mean_rm(y))
#> mean(x, na.rm = TRUE) + mean(y, na.rm = TRUE)
```

`!!` conserva la precedencia del operador porque funciona con expresiones.

```
x1 <- expr(x + 1)
x2 <- expr(x + 2)

expr(!!x1 / !!x2)
#> (x + 1)/(x + 2)
```



Si simplemente pegáramos el texto de las expresiones juntas, terminaríamos con `x + 1 / x + 2`, que tiene un AST muy diferente:



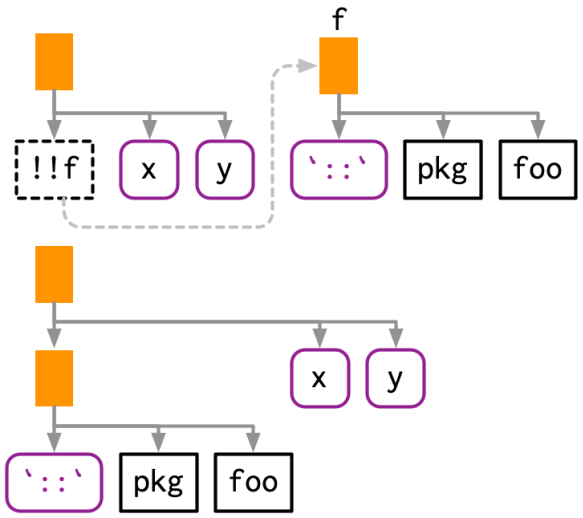
19.4.2. Remover citas de una función

`!!` se usa más comúnmente para reemplazar los argumentos de una función, pero también puede usarlo para reemplazar la función. El único desafío aquí es la precedencia de los operadores: `expr(!!f(x, y))` elimina las comillas del resultado de `f(x, y)`, por lo que necesita un par de paréntesis extra.

```
f <- expr(foo)
expr((!!f)(x, y))
#> foo(x, y)
```

Esto también funciona cuando `f` es una llamada:

```
f <- expr(pkg::foo)
expr((!!f)(x, y))
#> pkg::foo(x, y)
```



Debido a la gran cantidad de paréntesis involucrados, puede ser más claro usar `rlang::call2()`:

```
f <- expr(pkg::foo)
call2(f, expr(x), expr(y))
#> pkg::foo(x, y)
```

19.4.3. Remover cita de un argumento faltante

Muy ocasionalmente, es útil quitar las comillas de un argumento faltante (Section 18.6.2), pero el enfoque ingenuo no funciona:

```
arg <- missing_arg()
expr(foo(!!arg, !!arg))
#> Error in eval(expr, envir, enclos): argument "arg" is missing, with no default
```

Puedes solucionar esto con el ayudante `rlang::maybe_missing()`:

```
expr(foo(!!maybe_missing(arg), !!maybe_missing(arg)))
#> foo(, )
```

19.4.4. Remover cita de formas especiales

Hay algunas formas especiales en las que quitar las comillas es un error de sintaxis. Tome `$` por ejemplo: siempre debe ir seguido del nombre de una variable, no de otra expresión. Esto significa que intentar quitar las comillas con `$` fallará con un error de sintaxis:

```
expr(df$!!x)
#> Error: unexpected '!' in "expr(df$!"
```

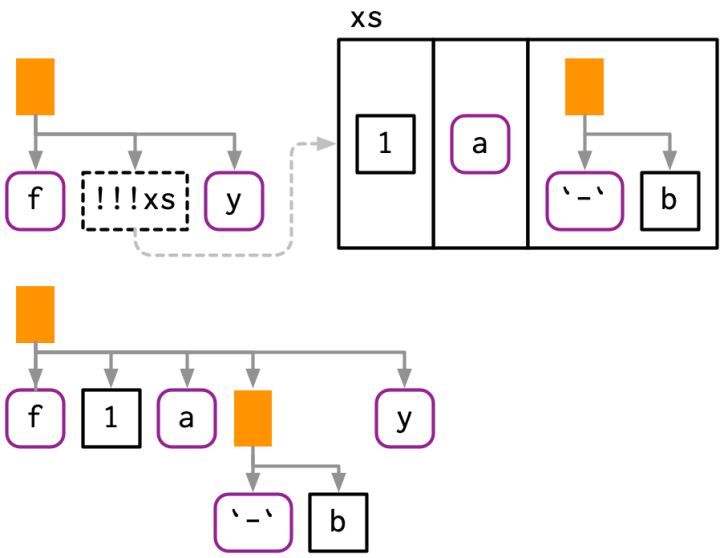
Para que la eliminación de comillas funcione, deberá usar la forma de prefijo (Section 6.8.1):

```
x <- expr(x)
expr(`$`(df, !!x))
#> df$x
```

19.4.5. Remover cita de muchos argumentos

`!!` es un reemplazo uno a uno. `!!!` (llamado "remover cita en conjunto", y pronunciado bang-bang-bang) es un reemplazo de uno a muchos. Toma una lista de expresiones y las inserta en la ubicación del `!!!`:

```
xs <- exprs(1, a, -b)
expr(f(!!!xs, y))
#> f(1, a, -b, y)

# O con nombres
ys <- set_names(xs, c("a", "b", "c"))
expr(f(!!!ys, d = 4))
#> f(a = 1, b = a, c = -b, d = 4)
```



`!!!` se puede usar en cualquier función rlang que tome `...` independientemente de si se cita o evalúa `...` o no. Volveremos a esto en la Section 19.6; por ahora tenga en cuenta que esto puede ser útil en `call2()`.

```
call2("f", !!!xs, expr(y))
#> f(1, a, -b, y)
```

19.4.6. La ficción educada de `!!`

Hasta ahora hemos actuado como si `!!` y `!!!` fueran operadores de prefijos regulares como `+`, `-` y `!`. No lo son. Desde la perspectiva de R, `!!` y `!!!` son simplemente la aplicación repetida de `!`:

```
!!TRUE
#> [1] TRUE
!!!TRUE
#> [1] FALSE
```

`!!` y `!!!` se comportan especialmente dentro de todas las funciones de citar impulsadas por rlang, donde se comportan como operadores reales con precedencia equivalente a `+` y `-` unarios. Esto requiere un trabajo considerable dentro de rlang, pero significa que puedes escribir `!!x + !!y` en lugar de `(!!x) + (!!y)`.

La mayor desventaja[3] de usar un operador falso es que puede obtener errores silenciosos al usar incorrectamente `!!` fuera de las funciones de cuasicitas. La mayoría de las veces esto no es un problema porque '!!' se usa típicamente para eliminar las comillas de expresiones o quóstulas. Dado que el operador de negación no admite expresiones, obtendrá un error de tipo de argumento en este caso:

```
x <- quote(variable)
!!x
#> Error in !x: invalid argument type
```

Pero puede obtener resultados incorrectos en silencio cuando trabaja con valores numéricos:

[3]Antes de R 3.5.1, había otra desventaja importante: el analizador de R trataba `!!x` como `!(!x)`. Esta es la razón por la que en las versiones antiguas de R es posible que vea paréntesis adicionales al imprimir expresiones. La buena noticia es que estos paréntesis no son reales y pueden ignorarse con seguridad la mayor parte del tiempo. La mala noticia es que se volverán reales si vuelves a analizar esa salida impresa en código R. Estas funciones de ida y vuelta no funcionarán como se espera, ya que `!(!x)` no elimina las comillas.

```
df <- data.frame(x = 1:5)
y <- 100
with(df, x + !!y)
#> [1] 2 3 4 5 6
```

Dados estos inconvenientes, es posible que se pregunte por qué introdujimos una nueva sintaxis en lugar de usar llamadas a funciones normales. De hecho, las primeras versiones de la evaluación ordenada usaban llamadas a funciones como `UQ()` y `UQS()`. Sin embargo, en realidad no son llamadas de función, y fingir que lo son conduce a un modo mental engañoso. Elegimos `!!` y `!!!` como la solución menos mala:

- Son visualmente fuertes y no se parecen a la sintaxis existente. Cuando vea `!!x` o `!!!x`, está claro que algo inusual está sucediendo.
- Anulan una parte de la sintaxis que rara vez se usa, ya que la doble negación no es un patrón común en R[4]. Si lo necesita, puede agregar paréntesis `!(!x)`.

19.4.7. AST no estándar

Sin comillas, es fácil crear AST no estándar, es decir, AST que contienen componentes que no son expresiones. (También es posible crear AST no estándar manipulando directamente los objetos subyacentes, pero es más difícil hacerlo accidentalmente). Estos son válidos y ocasionalmente útiles, pero su uso correcto está más allá del alcance de este libro. Sin embargo, es importante aprender acerca de ellos, porque se pueden dividir y, por lo tanto, imprimir de manera engañosa.

Por ejemplo, si alinea objetos más complejos, sus atributos no se imprimen. Esto puede conducir a resultados confusos:

[4] A diferencia de, por ejemplo, Javascript, donde `!!x` es un atajo de uso común para convertir un número entero en un número lógico.

```
x1 <- expr(class(!!data.frame(x = 10)))
x1
#> class(list(x = 10))
eval(x1)
#> [1] "data.frame"
```

Tienes dos herramientas principales para reducir esta confusión: `rlang::expr_print()` y `lobstr::ast()`:

```
expr_print(x1)
#> class(<df[,1]>)
lobstr::ast(!!x1)
#>  class
#>  <inline data.frame>
```

Otro caso confuso surge si alinea una secuencia de enteros:

```
x2 <- expr(f(!!c(1L, 2L, 3L, 4L, 5L)))
x2
#> f(1:5)
expr_print(x2)
#> f(<int: 1L, 2L, 3L, 4L, 5L>)
lobstr::ast(!!x2)
#>  f
#>  <inline integer>
```

También es posible crear AST regulares que no se pueden generar a partir del código debido a la precedencia del operador. En este caso, R imprimirá paréntesis que no existen en el AST:

```
x3 <- expr(1 + !!expr(2 + 3))
x3
#> 1 + (2 + 3)

lobstr::ast(!!x3)
#>  `+`
#>  1
#>   `+`
#>    2
#>    3
```

19.4.8. Ejercicios

1. Dados los siguientes componentes:

   ```
   xy <- expr(x + y)
   xz <- expr(x + z)
   yz <- expr(y + z)
   abc <- exprs(a, b, c)
   ```

   Utilice la cuasicita para construir las siguientes llamadas:

   ```
   (x + y) / (y + z)
   -(x + z) ^ (y + z)
   (x + y) + (y + z) - (x + y)
   atan2(x + y, y + z)
   sum(x + y, x + y, y + z)
   sum(a, b, c)
   mean(c(a, b, c), na.rm = TRUE)
   foo(a = x + y, b = y + z)
   ```

2. Las siguientes dos llamadas imprimen lo mismo, pero en realidad son diferentes:

```
(a <- expr(mean(1:10)))
#> mean(1:10)
(b <- expr(mean(!!(1:10))))
#> mean(1:10)
identical(a, b)
#> [1] FALSE
```

¿Cual es la diferencia? ¿Cuál es más natural?

19.5. No citar

Base R tiene una función que implementa la cuasicita: `bquote()`. Utiliza `.()` para quitar las comillas:

```
xyz <- bquote((x + y + z))
bquote(-.(xyz) / 2)
#> -(x + y + z)/2
```

`bquote()` no es utilizado por ninguna otra función en la base R y ha tenido un impacto relativamente pequeño en cómo se escribe el código R. Existen tres desafíos para el uso efectivo de `bquote()`:

- Solo se usa fácilmente con su código; es difícil aplicarlo a un código arbitrario proporcionado por un usuario.
- No proporciona un operador de empalme sin comillas que le permita quitar las comillas de varias expresiones almacenadas en una lista.
- Carece de la capacidad de manejar código acompañado de un entorno, lo cual es crucial para funciones que evalúan código en el contexto de un marco de datos, como `subset()` y amigos.

Las funciones base que citan un argumento usan alguna otra técnica para permitir la especificación indirecta. Los enfoques de Base R desactivan selectivamente las comillas, en lugar de usar la eliminación de comillas, por lo que las llamo técnicas **sin comillas**.

Hay cuatro formas básicas que se ven en la base R:

- Un par de funciones de citar y no citar. Por ejemplo, `$` tiene dos argumentos y el segundo argumento está entrecomillado. Esto es más fácil de ver si escribe en forma de prefijo: `mtcars$cyl` es equivalente a `` `$`(mtcars, cyl) ``. Si quiere referirse a una variable indirectamente, use `[[`, ya que toma el nombre de una variable como una cadena.

  ```
  x <- list(var = 1, y = 2)
  var <- "y"

  x$var
  #> [1] 1
  x[[var]]
  #> [1] 2
  ```

  Hay otras tres funciones de citar estrechamente relacionadas con `$`: `subset()`, `transform()` y `with()`. Estos se ven como envoltorios alrededor de `$` solo adecuados para uso interactivo, por lo que todos tienen la misma alternativa sin comillas: `[`

  `<-`/`assign()` y `::`/`getExportedValue()` work similarly to `$`/`[`.

- Un par de argumentos entre comillas y sin comillas. Por ejemplo, `rm()` le permite proporcionar nombres de variables simples en `...`, o un vector de caracteres de nombres de variables en `list`:

  ```
  x <- 1
  rm(x)

  y <- 2
  ```

```
vars <- c("y", "vars")
rm(list = vars)
```

`data()` y `save()` funcionan de forma similar.

- Un argumento que controla si un argumento diferente está entre comillas o no. Por ejemplo, en `library()`, el argumento `character.only` controla el comportamiento de las comillas del primer argumento, `package`:

```
library(MASS)

pkg <- "MASS"
library(pkg, character.only = TRUE)
```

`demo()`, `detach()`, `example()`, y `require()` funcionan de manera similar.

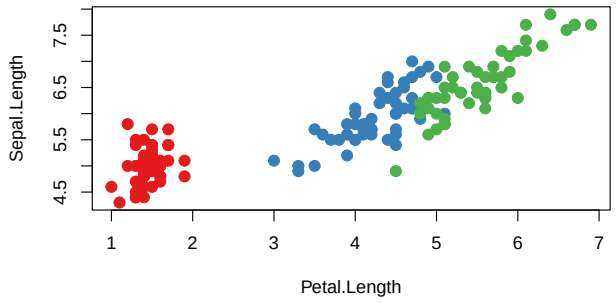
- Citar si falla la evaluación. Por ejemplo, el primer argumento de `help()` no está entrecomillado si se evalúa como una cadena; si la evaluación falla, se cita el primer argumento.

```
# Muestra ayuda para var
help(var)

var <- "mean"
# Muestra ayuda para la media
help(var)

var <- 10
# Muestra ayuda para var
help(var)
```

`ls()`, `page()`, y `match.fun()` funcionan de manera similar.

Otra clase importante de funciones de citar son las funciones básicas de modelado y trazado, que siguen las llamadas reglas de evaluación estándar no estándar: http://developer.r-project.org/nonstandard-eval.pdf. Por ejemplo, `lm()` cita los argumentos `weight` y `subset`, y cuando se usa con un argumento de fórmula, la función de trazado cita los argumentos estéticos (`col`, `cex`, etc.). Toma el siguiente código: solo necesitamos `col = Species` en lugar de `col = iris$Species`.

```
palette(RColorBrewer::brewer.pal(3, "Set1"))
plot(
  Sepal.Length ~ Petal.Length,
  data = iris,
  col = Species,
  pch = 20,
  cex = 2
)
```



Estas funciones no tienen opciones integradas para la especificación indirecta, pero aprenderá a simular la eliminación de comillas en la Section 20.6.

19.6. ... (dot-dot-dot)

`!!!` es útil porque no es raro tener una lista de expresiones que desea insertar en una llamada. Resulta que este patrón es común en otros lugares. Considere los siguientes dos problemas motivadores:

- ¿Qué haces si los elementos que quieres poner en `...` ya están almacenados en una lista? Por ejemplo, imagina que tienes una lista de data frames y deseas juntarlos con `rbind()`:

  ```
  dfs <- list(
    a = data.frame(x = 1, y = 2),
    b = data.frame(x = 3, y = 4)
  )
  ```

  Podrías resolver este caso específico con `rbind(dfs$a, dfs$b)`, pero ¿cómo generalizas esa solución a una lista de longitud arbitraria?

- ¿Qué hace si desea proporcionar el nombre del argumento indirectamente? Por ejemplo, imagine que desea crear un marco de datos de una sola columna donde el nombre de la columna se especifica en una variable:

  ```
  var <- "x"
  val <- c(4, 3, 9)
  ```

  En este caso, podría crear un marco de datos y luego cambiar los nombres (es decir, `setNames(data.frame(val), var)`), pero esto parece poco elegante. ¿Cómo podemos hacerlo mejor?

Una forma de pensar en estos problemas es trazar paralelos explícitos con la cuasicitar:

- La vinculación de filas de varios marcos de datos es como el empalme sin comillas: queremos incorporar elementos individuales de la lista en la llamada:

```
dplyr::bind_rows(!!!dfs)
#>   x y
#> 1 1 2
#> 2 3 4
```

Cuando se usa en este contexto, el comportamiento de `!!!` se conoce como "salpicar" en Ruby, Go, PHP y Julia. Está estrechamente relacionado con `*args` (star-args) y `**kwarg` (star-star-kwargs) en Python, que a veces se denomina desempaquetado de argumentos.

- El segundo problema es como quitar las comillas del lado izquierdo de `=`: en lugar de interpretar literalmente `var`, queremos usar el valor almacenado en la variable llamada `var`:

```
tibble::tibble(!!var := val)
#> # A tibble: 3 × 1
#>       x
#>   <dbl>
#> 1     4
#> 2     3
#> 3     9
```

Tenga en cuenta el uso de `:=` (pronunciado dos puntos-igual) en lugar de `=`. Desafortunadamente, necesitamos esta nueva operación porque la gramática de R no permite expresiones como nombres de argumentos:

```
tibble::tibble(!!var = value)
#> Error: unexpected '=' in "tibble::tibble(!!var ="
```

`:=` es como un órgano vestigial: es reconocido por el analizador de R, pero no tiene ningún código asociado. Parece un `=` pero permite expresiones en ambos lados, lo que lo convierte en una alternativa más flexible que `=`. Se usa en data.table por razones similares.

Base R adopta un enfoque diferente, al que volveremos en la Section 19.6.4.

Decimos que las funciones que admiten estas herramientas, sin citar argumentos, tienen **puntos ordenados**[5]. Para obtener un comportamiento de puntos ordenados en su propia función, todo lo que necesita hacer es usar `list2()`.

19.6.1. Ejemplos

Un lugar en el que podríamos usar `list2()` es crear un contenedor alrededor de `attributes()` que nos permita establecer atributos de manera flexible:

```
set_attr <- function(.x, ...) {
  attr <- rlang::list2(...)
  attributes(.x) <- attr
  .x
}

attrs <- list(x = 1, y = 2)
attr_name <- "z"

1:10 %>%
  set_attr(w = 0, !!!attrs, !!attr_name := 3) %>%
  str()
#>  int [1:10] 1 2 3 4 5 6 7 8 9 10
#>  - attr(*, "w")= num 0
#>  - attr(*, "x")= num 1
#>  - attr(*, "y")= num 2
#>  - attr(*, "z")= num 3
```

[5]Es cierto que este no es el nombre más creativo, pero sugiere claramente que es algo que se agregó a R después del hecho.

19.6.2. `exec()`

¿Qué sucede si desea utilizar esta técnica con una función que no tiene puntos ordenados? Una opción es usar `rlang::exec()` para llamar a una función con algunos argumentos suministrados directamente (en `...`) y otros indirectamente (en una lista):

```
# Directamente
exec("mean", x = 1:10, na.rm = TRUE, trim = 0.1)
#> [1] 5.5

# Indirectamente
args <- list(x = 1:10, na.rm = TRUE, trim = 0.1)
exec("mean", !!!args)
#> [1] 5.5

# Ambos
params <- list(na.rm = TRUE, trim = 0.1)
exec("mean", x = 1:10, !!!params)
#> [1] 5.5
```

`rlang::exec()` también hace posible proporcionar nombres de argumentos indirectamente:

```
arg_name <- "na.rm"
arg_val <- TRUE
exec("mean", 1:10, !!arg_name := arg_val)
#> [1] 5.5
```

Y finalmente, es útil si tiene un vector de nombres de funciones o una lista de funciones que desea llamar con los mismos argumentos:

```
x <- c(runif(10), NA)
funs <- c("mean", "median", "sd")

purrr::map_dbl(funs, exec, x, na.rm = TRUE)
#> [1] 0.444 0.482 0.298
```

`exec()` está estrechamente relacionado con `call2()`; donde `call2()` devuelve una expresión, `exec()` la evalúa.

19.6.3. `dots_list()`

`list2()` proporciona otra característica útil: de forma predeterminada, ignorará cualquier argumento vacío al final. Esto es útil en funciones como `tibble::tibble()` porque significa que puedes cambiar fácilmente el orden de las variables sin preocuparte por la coma final:

```
# Puede mover fácilmente x a la primera entrada:
tibble::tibble(
  y = 1:5,
  z = 3:-1,
  x = 5:1,
)

# Necesita eliminar la coma de z y agregar una coma a x
data.frame(
  y = 1:5,
  z = 3:-1,
  x = 5:1
)
```

`list2()` es un envoltorio alrededor de `rlang::dots_list()` con los valores predeterminados establecidos en las configuraciones más utilizadas. Puedes tener más control llamando a `dots_list()` directamente:

- `.ignore_empty` le permite controlar exactamente qué argumentos se ignoran. El valor predeterminado ignora un solo argumento final para obtener el comportamiento descrito anteriormente, pero puede optar por ignorar todos los argumentos faltantes o ninguno.

- `.homonyms` controla lo que sucede si varios argumentos usan el mismo nombre:

```
str(dots_list(x = 1, x = 2))
#> List of 2
#>  $ x: num 1
#>  $ x: num 2
str(dots_list(x = 1, x = 2, .homonyms = "first"))
#> List of 1
#>  $ x: num 1
str(dots_list(x = 1, x = 2, .homonyms = "last"))
#> List of 1
#>  $ x: num 2
str(dots_list(x = 1, x = 2, .homonyms = "error"))
#> Error:
#> ! Arguments in `...` must have unique names.
#>  Multiple arguments named `x` at positions 1 and 2.
```

- Si hay argumentos vacíos que no se ignoran, `.preserve_empty` controla qué hacer con ellos. El valor predeterminado arroja un error; establecer `.preserve_empty = TRUE` en su lugar devuelve los símbolos que faltan. Esto es útil si estás usando `dots_list()` para generar llamadas a funciones.

19.6.4. Con R base

Base R proporciona una navaja suiza para resolver estos problemas: `do.call()`. `do.call()` tiene dos argumentos principales. El primer argumento, `qué`, da una función para llamar. El segundo argumento,

`args`, es una lista de argumentos para pasar a esa función, por lo que `do.call("f", list(x, y, z))` es equivalente a `f(x, y, z)`.

- `do.call()` da una solución directa a `rbind()` juntando muchos marcos de datos:

```
do.call("rbind", dfs)
#>   x y
#> a 1 2
#> b 3 4
```

- Con un poco más de trabajo, podemos usar `do.call()` para resolver el segundo problema. Primero creamos una lista de argumentos, luego le damos un nombre y luego usamos `do.call()`:

```
args <- list(val)
names(args) <- var

do.call("data.frame", args)
#>   x
#> 1 4
#> 2 3
#> 3 9
```

Algunas funciones básicas (incluidas `interaction()`, `expand.grid()`, `options()` y `par()`) usan un truco para evitar `do.call()`: si el primer componente de `...` es una lista, tomarán sus componentes en lugar de mirar los otros elementos de `...`. La implementación se parece a esto:

```
f <- function(...) {
  dots <- list(...)
  if (length(dots) == 1 && is.list(dots[[1]])) {
    dots <- dots[[1]]
  }
```

```
  # Hacer algo
  ...
}
```

Otro método para evitar `do.call()` se encuentra en la función `RCurl::getURL()` escrita por Duncan Temple Lang. `getURL()` toma tanto `...` como `.dots`, que se concatenan entre sí y se parece a esto:

```
f <- function(..., .dots) {
  dots <- c(list(...), .dots)
  # Hacer algo
}
```

En el momento en que lo descubrí, encontré esta técnica particularmente convincente, por lo que puede verla utilizada en todo el tidyverse. Ahora, sin embargo, prefiero el enfoque descrito anteriormente.

19.6.5. Ejercicios

1. A continuación se muestra una forma de implementar `exec()`. Describa cómo funciona. ¿Cuáles son las ideas clave?

   ```
   exec <- function(f, ..., .env = caller_env()) {
     args <- list2(...)
     do.call(f, args, envir = .env)
   }
   ```

2. Lea atentamente el código fuente de `interaction()`, `expand.grid()` y `par()`. Compare y contraste las técnicas que usan para cambiar entre puntos y comportamiento de lista.

3. Explique el problema con esta definición de `set_attr()`

```
set_attr <- function(x, ...) {
  attr <- rlang::list2(...)
  attributes(x) <- attr
  x
}
set_attr(1:10, x = 10)
#> Error in attributes(x) <- attr: attributes must be named
```

19.7. Casos de estudio

Para concretar las ideas de la cuasicitación, esta sección contiene algunos pequeños estudios de casos que la utilizan para resolver problemas reales. Algunos de los estudios de casos también usan purrr: encuentro que la combinación de cuasicitación y programación funcional es particularmente elegante.

19.7.1. `lobstr::ast()`

La cuasicita nos permite resolver un molesto problema con `lobstr::ast()`: ¿qué sucede si ya capturamos la expresión?

```
z <- expr(foo(x, y))
lobstr::ast(z)
#> z
```

Debido a que `ast()` cita su primer argumento, podemos usar `!!`:

```
lobstr::ast(!!z)
#>  foo
#>  x
#>  y
```

19.7.2. Map-reduce para generar código

Quasiquotation nos brinda herramientas poderosas para generar código, particularmente cuando se combina con `purrr::map()` y `purrr::reduce()`. Por ejemplo, suponga que tiene un modelo lineal especificado por los siguientes coeficientes:

```
intercept <- 10
coefs <- c(x1 = 5, x2 = -4)
```

Y quieres convertirlo en una expresión como `10 + (x1 * 5) + (x2 * -4)`. Lo primero que debemos hacer es convertir el vector de nombres de personajes en una lista de símbolos. `rlang::syms()` está diseñado precisamente para este caso:

```
coef_sym <- syms(names(coefs))
coef_sym
#> [[1]]
#> x1
#>
#> [[2]]
#> x2
```

A continuación, debemos combinar cada nombre de variable con su coeficiente. Podemos hacer esto combinando `rlang::expr()` con `purrr::map2()`:

```
summands <- map2(coef_sym, coefs, ~ expr((!!.x * !!.y)))
summands
#> [[1]]
#> (x1 * 5)
#>
#> [[2]]
#> (x2 * -4)
```

En este caso, el intercepto también es parte de la suma, aunque no implica una multiplicación. Simplemente podemos agregarlo al comienzo del vector `summands`:

```
summands <- c(intercept, summands)
summands
#> [[1]]
#> [1] 10
#>
#> [[2]]
#> (x1 * 5)
#>
#> [[3]]
#> (x2 * -4)
```

Finalmente, necesitamos reducir (Section 9.5) los términos individuales en una sola suma sumando las piezas:

```
eq <- reduce(summands, ~ expr(!!.x + !!.y))
eq
#> 10 + (x1 * 5) + (x2 * -4)
```

Podríamos hacer esto aún más general al permitir que el usuario proporcione el nombre del coeficiente y, en lugar de asumir muchas variables diferentes, indexar en una sola.

```
var <- expr(y)
coef_sym <- map(seq_along(coefs), ~ expr((!!var)[[!!.x]]))
coef_sym
#> [[1]]
#> y[[1L]]
#>
#> [[2]]
#> y[[2L]]
```

Y termine envolviendo esto en una función:

```
linear <- function(var, val) {
  var <- ensym(var)
  coef_name <- map(seq_along(val[-1]), ~ expr((!!var)[[!!.x]]))

  summands <- map2(val[-1], coef_name, ~ expr((!!.x * !!.y)))
  summands <- c(val[[1]], summands)

  reduce(summands, ~ expr(!!.x + !!.y))
}

linear(x, c(10, 5, -4))
#> 10 + (5 * x[[1L]]) + (-4 * x[[2L]])
```

Tenga en cuenta el uso de `ensym()`: queremos que el usuario proporcione el nombre de una sola variable, no una expresión más compleja.

19.7.3. Cortar un arreglo

Una herramienta ocasionalmente útil que falta en la base R es la capacidad de extraer una porción de una matriz dada una dimensión y un índice. Por ejemplo, nos gustaría escribir `slice(x, 2, 1)` para extraer el primer segmento a lo largo de la segunda dimensión, es decir, `x[, 1, ]`. Este es un problema moderadamente desafiante porque requiere trabajar con argumentos faltantes.

Tendremos que generar una llamada con varios argumentos faltantes. Primero generamos una lista de argumentos faltantes con `rep()` y `missing_arg()`, luego quitamos las comillas y los empalmamos en una llamada:

```
indices <- rep(list(missing_arg()), 3)
expr(x[!!!indices])
#> x[, , ]
```

Luego usamos la asignación de subconjuntos para insertar el índice en la posición deseada:

```
indices[[2]] <- 1
expr(x[!!!indices])
#> x[, 1, ]
```

Luego envolvemos esto en una función, usando un par de `stopifnot()`s para aclarar la interfaz:

```
slice <- function(x, along, index) {
  stopifnot(length(along) == 1)
  stopifnot(length(index) == 1)

  nd <- length(dim(x))
  indices <- rep(list(missing_arg()), nd)
  indices[[along]] <- index

  expr(x[!!!indices])
}

x <- array(sample(30), c(5, 2, 3))
slice(x, 1, 3)
#> x[3, , ]
slice(x, 2, 2)
#> x[, 2, ]
slice(x, 3, 1)
#> x[, , 1]
```

Un `slice()` real evaluaría la llamada generada (Chapter 20), pero aquí creo que es más esclarecedor ver el código que se genera, ya que esa es la parte difícil del desafío.

19.7.4. Creación de funciones

Otra poderosa aplicación de las citas es la creación de funciones "a mano", usando `rlang::new_function()`. Es una función que crea una función a partir de sus tres componentes (Section 6.2.1): argumentos, cuerpo y (opcionalmente) un entorno:

```
new_function(
  exprs(x = , y = ),
  expr({x + y})
)
#> function (x, y)
#> {
#>     x + y
#> }
```

NB: Los argumentos vacíos en `exprs()` generan argumentos sin valores predeterminados.

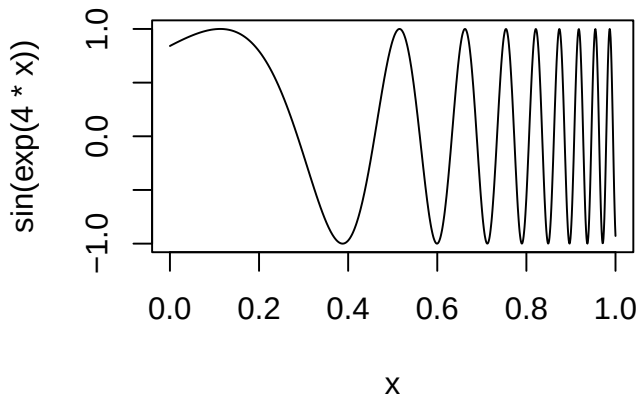
Un uso de `new_function()` es como una alternativa a las fábricas de funciones con argumentos escalares o de símbolos. Por ejemplo, podríamos escribir una función que genere funciones que eleven una función a la potencia de un número.

```
power <- function(exponent) {
  new_function(
    exprs(x = ),
    expr({
      x ^ !!exponent
```

```
    }),
    caller_env()
  )
}
power(0.5)
#> function (x)
#> {
#>     x^0.5
#> }
```

Otra aplicación de `new_function()` es para funciones que funcionan como `graphics::curve()`, que te permite trazar una expresión matemática sin crear una función:

```
curve(sin(exp(4 * x)), n = 1000)
```



En este código, `x` es un pronombre: no representa un solo valor concreto, sino que es un marcador de posición que varía en el rango de la gráfica. Una forma de implementar `curve()` es convertir esa expresión en una función con un solo argumento, `x`, y luego llamar a esa función:

```
curve2 <- function(expr, xlim = c(0, 1), n = 100) {
  expr <- enexpr(expr)
  f <- new_function(exprs(x = ), expr)

  x <- seq(xlim[1], xlim[2], length = n)
  y <- f(x)

  plot(x, y, type = "l", ylab = expr_text(expr))
}
curve2(sin(exp(4 * x)), n = 1000)
```

Las funciones como `curve()` que usan una expresión que contiene un pronombre se conocen como funciones **anafóricas**[6].

19.7.5. Ejercicios

1. En el ejemplo del modelo lineal, podríamos reemplazar `expr()` en `reduce(summands, ~ expr(!!.x + !!.y))` con `call2()`: `reduce(summands, call2 , "+")`. Compare y contraste los dos enfoques. ¿Cuál crees que es más fácil de leer?

2. Vuelva a implementar la transformación de Box-Cox definida a continuación usando la eliminación de comillas y `new_function()`:

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

[6] Anaphoric proviene del término lingüístico "anáfora", una expresión que depende del contexto. Las funciones anafóricas se encuentran en Arc (un lenguaje parecido a Lisp), Perl, y Clojure.

```
  }
}
```

3. Vuelva a implementar el `compose()` simple definido a continuación usando quasiquotation y `new_function()`:

```
compose <- function(f, g) {
  function(...) f(g(...))
}
```

19.8. Historia

La idea de la cuasicita es antigua. Fue desarrollado por primera vez por el filósofo Willard van Orman Quine[7] a principios de la década de 1940. Es necesario en filosofía porque ayuda a delinear con precisión el uso y la mención de palabras, es decir, distinguir entre el objeto y las palabras que usamos para referirnos a ese objeto.

Quasiquotation se utilizó por primera vez en un lenguaje de programación, Lisp, a mediados de la década de 1970 (Bawden 1999). Lisp tiene una función de comillas `` ` ``, y usa `,` para quitar las comillas. La mayoría de los lenguajes con herencia Lisp se comportan de manera similar. Por ejemplo, Racket (`` ` `` y `@`), Clojure (`` ` `` y `~`) y Julia (`:` y `@`) tienen herramientas de cuasicitación que difieren solo ligeramente de Lisp . Estos idiomas tienen una función de comillas simples y debe llamarla explícitamente.

En R, sin embargo, muchas funciones citan una o más entradas. Esto introduce ambigüedad (porque necesita leer la documentación para determinar si un argumento se cita o no), pero permite un código de exploración de datos conciso y elegante. En base R, solo una función admite la cuasicita: `bquote()`, escrita en 2003 por Thomas Lumley. Sin embargo, `bquote()`

[7] Es posible que esté familiarizado con el nombre Quine de "quines", programas de computadora que devuelven una copia de su propia fuente cuando se ejecutan.

tiene algunas limitaciones importantes que le impidieron tener un gran impacto en el código R (Section 19.5).

Mi intento de resolver estas limitaciones condujo al paquete lazyeval (2014-2015). Desafortunadamente, mi análisis del problema estaba incompleto y aunque lazyeval resolvió algunos problemas, creó otros. No fue hasta que comencé a trabajar con Lionel Henry en el problema que todas las piezas finalmente encajaron y creamos el marco de evaluación ordenado completo (2017). A pesar de la novedad de la evaluación ordenada, la enseño aquí porque es una teoría rica y poderosa que, una vez dominada, hace que muchos problemas difíciles sean mucho más fáciles.

20. Evaluación

20.1. Introducción

El inverso de la cita de cara al usuario es la no cita: le da al *usuario* la capacidad de evaluar selectivamente partes de un argumento citado de otro modo. El complemento de cotización para el desarrollador es la evaluación: esto le da al *desarrollador* la capacidad de evaluar expresiones citadas en entornos personalizados para lograr objetivos específicos.

Este capítulo comienza con una discusión de la evaluación en su forma más pura. Aprenderá cómo `eval()` evalúa una expresión en un entorno, y luego cómo se puede usar para implementar una serie de importantes funciones básicas de R. Una vez que tenga los conceptos básicos bajo su cinturón, aprenderá las extensiones a la evaluación que se necesitan para la solidez. Hay dos grandes ideas nuevas:

- El quosure: una estructura de datos que captura una expresión junto con su entorno asociado, como se encuentra en los argumentos de función.
- La máscara de datos, que facilita la evaluación de una expresión en el contexto de un marco de datos. Esto introduce una posible ambigüedad de evaluación que luego resolveremos con pronombres de datos.

Juntos, la cuasicotización, las cuotas y las máscaras de datos forman lo que llamamos **evaluación ordenada**, o evaluación ordenada para abreviar. Tidy eval proporciona un enfoque basado en principios para la evaluación

no estándar que hace posible el uso de dichas funciones de forma interactiva e integrada con otras funciones. La evaluación ordenada es la implicación práctica más importante de toda esta teoría, por lo que dedicaremos un poco de tiempo a explorar las implicaciones. El capítulo termina con una discusión de los enfoques relacionados más cercanos en base R, y cómo puede programar alrededor de sus inconvenientes.

Estructura

- La Section 20.2 analiza los aspectos básicos de la evaluación usando `eval()` y muestra cómo puede usarlo para implementar funciones clave como `local()` y `source()`.
- La Section 20.3 introduce una nueva estructura de datos, el quosure, que combina una expresión con un entorno. Aprenderá cómo capturar cuotas de promesas y evaluarlas usando `rlang::eval_tidy()`.
- La Section 20.4 extiende la evaluación con la máscara de datos, lo que hace que sea trivial entremezclar símbolos vinculados en un entorno con variables que se encuentran en un marco de datos.
- La Section 20.5 muestra cómo usar la evaluación ordenada en la práctica, centrándose en el patrón común de citar y quitar las comillas, y cómo manejar la ambigüedad con los pronombres.
- La Section 20.6 vuelve a la evaluación en base R, analiza algunas de las desventajas y muestra cómo usar la cuasicita y la evaluación para envolver funciones que usan NSE.

Requisitos previos

Deberá estar familiarizado con el contenido del Chapter 18 y el Chapter 19, así como con la estructura de datos del entorno (Section 7.2) y el entorno de la persona que llama (Section 7.5).

Seguiremos usando rlang y purrr.

```
library(rlang)
library(purrr)
```

20.2. Conceptos básicos de evaluación

Aquí exploraremos los detalles de `eval()` que mencionamos brevemente en el último capítulo. Tiene dos argumentos clave: `expr` y `env`. El primer argumento, `expr`, es el objeto a evaluar, típicamente un símbolo o expresión[1]. Ninguna de las funciones de evaluación cita sus entradas, por lo que normalmente las usará con `expr()` o similar:

```
x <- 10
eval(expr(x))
#> [1] 10

y <- 2
eval(expr(x + y))
#> [1] 12
```

El segundo argumento, `env`, proporciona el entorno en el que debe evaluarse la expresión, es decir, dónde buscar los valores de `x`, `y` y `+`. De forma predeterminada, este es el entorno actual, es decir, el entorno de llamada de `eval()`, pero puede anularlo si lo desea:

```
eval(expr(x + y), env(x = 1000))
#> [1] 1002
```

[1]Todos los demás objetos se rinden cuando se evalúan; es decir, `eval(x)` produce `x`, excepto cuando `x` es un símbolo o una expresión.

El primer argumento se evalúa, no se cita, lo que puede generar resultados confusos una vez si usa un entorno personalizado y se olvida de citar manualmente:

```
eval(print(x + 1), env(x = 1000))
#> [1] 11
#> [1] 11

eval(expr(print(x + 1)), env(x = 1000))
#> [1] 1001
```

Ahora que ha visto los conceptos básicos, exploremos algunas aplicaciones. Nos centraremos principalmente en las funciones básicas de R que podría haber usado antes, volviendo a implementar los principios subyacentes usando rlang.

20.2.1. Aplicar: `local()`

A veces, desea realizar una parte del cálculo que crea algunas variables intermedias. Las variables intermedias no tienen un uso a largo plazo y podrían ser bastante grandes, por lo que preferiría no mantenerlas. Un enfoque es limpiar lo que ensucia usando `rm()`; otra es envolver el código en una función y simplemente llamarlo una vez. Un enfoque más elegante es usar `local()`:

```
# Limpiar variables creadas anteriormente
rm(x, y)

foo <- local({
  x <- 10
  y <- 200
  x + y
```

```
})

foo
#> [1] 210
x
#> Error in eval(expr, envir, enclos): object 'x' not found
y
#> Error in eval(expr, envir, enclos): object 'y' not found
```

La esencia de `local()` es bastante simple y se vuelve a implementar a continuación. Capturamos la expresión de entrada y creamos un nuevo entorno en el que evaluarla. Este es un nuevo entorno (por lo que la asignación no afecta el entorno existente) con el entorno de la persona que llama como padre (para que `expr` aún pueda acceder a las variables en ese entorno). Esto emula efectivamente la ejecución de `expr` como si estuviera dentro de una función (es decir, tiene un alcance léxico, Section 6.4).

```
local2 <- function(expr) {
  env <- env(caller_env())
  eval(enexpr(expr), env)
}

foo <- local2({
  x <- 10
  y <- 200
  x + y
})

foo
#> [1] 210
x
#> Error in eval(expr, envir, enclos): object 'x' not found
```

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

Comprender cómo funciona `base::local()` es más difícil, ya que usa `eval()` y `substitute()` juntos de formas bastante complicadas. Descubrir exactamente lo que está pasando es una buena práctica si realmente quiere comprender las sutilezas de `substitute()` y las funciones base `eval()`, por lo que se incluyen en los ejercicios a continuación.

20.2.2. Aplicar: `source()`

Podemos crear una versión simple de `source()` combinando `eval()` con `parse_expr()` de la Section 18.4.3. Leemos el archivo desde el disco, usamos `parse_expr()` para analizar la cadena en una lista de expresiones y luego usamos `eval()` para evaluar cada elemento a su vez. Esta versión evalúa el código en el entorno de la persona que llama e invisiblemente devuelve el resultado de la última expresión en el archivo como `base::source()`.

```
source2 <- function(path, env = caller_env()) {
  file <- paste(readLines(path, warn = FALSE), collapse = "\n")
  exprs <- parse_exprs(file)

  res <- NULL
  for (i in seq_along(exprs)) {
    res <- eval(exprs[[i]], env)
  }

  invisible(res)
}
```

La 'source()' real es considerablemente más complicada porque puede hacer `eco` de entrada y salida, y tiene muchas otras configuraciones que controlan su comportamiento.

Vectores de expresión

`base::eval()` tiene un comportamiento especial para la expresión *vectores*, evaluando cada componente a su vez. Esto lo convierte en una implementación muy compacta de `source2()` porque `base::parse()` también devuelve un objeto de expresión:

```
source3 <- function(file, env = parent.frame()) {
  lines <- parse(file)
  res <- eval(lines, envir = env)
  invisible(res)
}
```

Mientras que `source3()` es considerablemente más conciso que `source2()`, esta es la única ventaja de los vectores de expresión. En general, no creo que este beneficio compense el costo de introducir una nueva estructura de datos y, por lo tanto, este libro evita el uso de vectores de expresión.

20.2.3. Gotcha: `function()`

Hay un pequeño inconveniente que debe tener en cuenta si está utilizando `eval()` y `expr()` para generar funciones:

```
x <- 10
y <- 20
f <- eval(expr(function(x, y) !!x + !!y))
f
#> function(x, y) !!x + !!y
```

Esta función no parece funcionar, pero lo hace:

```
f()
#> [1] 30
```

Esto se debe a que, si está disponible, las funciones imprimen su atributo `srcref` (Section 6.2.1), y debido a que `srcref` es una característica base de R, no reconoce las cuasicita.

Para solucionar este problema, utilice `new_function()` (Section 19.7.4) o elimine el atributo `srcref`:

```
attr(f, "srcref") <- NULL
f
#> function (x, y)
#> 10 + 20
```

20.2.4. Ejercicios

1. Lea atentamente la documentación de `source()`. ¿Qué entorno usa por defecto? ¿Qué sucede si proporciona `local = TRUE`? ¿Cómo se proporciona un entorno personalizado?

2. Prediga los resultados de las siguientes líneas de código:

   ```
   eval(expr(eval(expr(eval(expr(2 + 2))))))
   eval(eval(expr(eval(expr(eval(expr(2 + 2)))))))
   expr(eval(expr(eval(expr(eval(expr(2 + 2)))))))
   ```

3. Complete los cuerpos de función a continuación para volver a implementar `get()` usando `sym()` y `eval()`, y `assign()` usando `sym()`, `expr()` y `eval ()`. No se preocupe por las múltiples formas de elegir un entorno que admiten `get()` y `assign()`; suponga que el usuario lo proporciona explícitamente.

```
# el nombre es una cadena
get2 <- function(name, env) {}
assign2 <- function(name, value, env) {}
```

4. Modifique `source2()` para que devuelva el resultado de *todas* las expresiones, no solo la última. ¿Puedes eliminar el bucle for?

5. Podemos hacer que `base::local()` sea un poco más fácil de entender al distribuirlo en varias líneas:

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

   Explique cómo funciona `local()` en palabras. (Sugerencia: es posible que desee `print(call)` para ayudar a comprender qué está haciendo `substitute()`, y lea la documentación para recordar de qué entorno se heredará `new.env()`).

20.3. Quosures

Casi todos los usos de `eval()` implican tanto una expresión como un entorno. Este acoplamiento es tan importante que necesitamos una estructura de datos que pueda contener ambas piezas. Base R no tiene tal estructura [2], por lo que rlang llena el espacio con **quosure**, un objeto que contiene una expresión y un entorno. El nombre es un acrónimo de cita y cierre, porque un quosure cita la expresión y encierra el entorno. Quosures cosifica el objeto de promesa interna (Section 6.5.1) en algo con lo que puede programar.

[2] Técnicamente, una fórmula combina una expresión y un entorno, pero las fórmulas están estrechamente vinculadas al modelado, por lo que una nueva estructura de datos tiene sentido.

En esta sección, aprenderá cómo crear y manipular quosures, y un poco sobre cómo se implementan.

20.3.1. Creando

Hay tres formas de crear quosures:

- Use `enquo()` y `enquos()` para capturar expresiones proporcionadas por el usuario. La gran mayoría de las cuotas deben crearse de esta manera.

```
foo <- function(x) enquo(x)
foo(a + b)
#> <quosure>
#> expr: ^a + b
#> env:  global
```

- `quo()` y `quos()` existen para coincidir con `expr()` y `exprs()`, pero se incluyen solo para completar y se necesitan muy raramente. Si te encuentras usándolos, piensa cuidadosamente si `expr()` y quitar las comillas cuidadosamente pueden eliminar la necesidad de capturar el entorno.

```
quo(x + y + z)
#> <quosure>
#> expr: ^x + y + z
#> env:  global
```

- `new_quosure()` crear un quosure a partir de sus componentes: una expresión y un entorno. Esto rara vez se necesita en la práctica, pero es útil para el aprendizaje, por lo que se usa mucho en este capítulo.

```
new_quosure(expr(x + y), env(x = 1, y = 10))
#> <quosure>
```

```
#> expr: ^x + y
#> env:  0x556a66e44be8
```

20.3.2. Evaluando

Las cuotas se combinan con una nueva función de evaluación `eval_tidy()` que toma una única cuota en lugar de un par expresión-entorno. Es fácil de usar:

```
q1 <- new_quosure(expr(x + y), env(x = 1, y = 10))
eval_tidy(q1)
#> [1] 11
```

Para este caso simple, `eval_tidy(q1)` es básicamente un atajo para `eval(get_expr(q1), get_env(q1))`. Sin embargo, tiene dos características importantes de las que aprenderá más adelante en el capítulo: admite quósures anidados (Section 20.3.5) y pronombres (Section 20.4.2).

20.3.3. Puntos

Los quosures suelen ser solo una conveniencia: hacen que el código sea más limpio porque solo tiene un objeto para pasar, en lugar de dos. Sin embargo, son esenciales cuando se trata de trabajar con `...` porque es posible que cada argumento pasado a `...` se asocie con un entorno diferente. En el siguiente ejemplo, tenga en cuenta que ambos quosures tienen la misma expresión, `x`, pero un entorno diferente:

```
f <- function(...) {
  x <- 1
  g(..., f = x)
}
```

```
g <- function(...) {
  enquos(...)
}

x <- 0
qs <- f(global = x)
qs
#> <list_of<quosure>>
#>
#> $global
#> <quosure>
#> expr: ^x
#> env:  global
#>
#> $f
#> <quosure>
#> expr: ^x
#> env:  0x556a6a2c3758
```

Eso significa que cuando los evalúas, obtienes los resultados correctos:

```
map_dbl(qs, eval_tidy)
#> global      f
#>      0      1
```

Evaluar correctamente los elementos de `...` fue una de las motivaciones originales para el desarrollo de quosures.

20.3.4. Bajo el capó

Quosures se inspiró en las fórmulas de R, porque las fórmulas capturan una expresión y un entorno:

```
f <- ~runif(3)
str(f)
#> Class 'formula'  language ~runif(3)
#>   ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

Una versión anterior de la evaluación ordenada usaba fórmulas en lugar de quosures, ya que una característica atractiva de ~ es que proporciona citas con una sola pulsación de tecla. Desafortunadamente, sin embargo, no hay una forma limpia de hacer que ~ sea una función de cuasicomillas.

Las cuotas son una subclase de fórmulas:

```
q4 <- new_quosure(expr(x + y + z))
class(q4)
#> [1] "quosure" "formula"
```

lo que significa que, bajo el capó, las coyunturas, como las fórmulas, son objetos de llamada:

```
is_call(q4)
#> [1] TRUE

q4[[1]]
#> Warning: Subsetting quosures with `[[` is deprecated as of rlang 0.4.0
#> Please use `quo_get_expr()` instead.
#> This warning is displayed once every 8 hours.
#> `~`
q4[[2]]
#> x + y + z
```

con un atributo que almacena el entorno:

```
attr(q4, ".Environment")
#> <environment: R_GlobalEnv>
```

Si necesita extraer la expresión o el entorno, no confíe en estos detalles de implementación. En su lugar, utilice `get_expr()` y `get_env()`:

```
get_expr(q4)
#> x + y + z
get_env(q4)
#> <environment: R_GlobalEnv>
```

20.3.5. Cuosuras anidadas

Es posible usar cuasicomillas para incrustar una quosure en una expresión. Esta es una herramienta avanzada, y la mayoría de las veces no necesita pensar en ella porque simplemente funciona, pero hablo de ella aquí para que pueda detectar quosures anidados en la naturaleza y no confundirse. Tome este ejemplo, que alinea dos quosures en una expresión:

```
q2 <- new_quosure(expr(x), env(x = 1))
q3 <- new_quosure(expr(x), env(x = 10))

x <- expr(!!q2 + !!q3)
```

Se evalúa correctamente con `eval_tidy()`:

```
eval_tidy(x)
#> [1] 11
```

Sin embargo, si lo imprime, solo verá las 'x', con la herencia de su fórmula filtrándose:

```
x
#> (~x) + ~x
```

Puede obtener una mejor visualización con `rlang::expr_print()` (Section 19.4.7):

```
expr_print(x)
#> (^x) + (^x)
```

Cuando usa `expr_print()` en la consola, los quosures se colorean de acuerdo con su entorno, lo que facilita detectar cuándo los símbolos están vinculados a diferentes variables.

20.3.6. Ejercicios

1. Predecir lo que devolverá cada uno de los siguientes quosures si se evalúan.

```
q1 <- new_quosure(expr(x), env(x = 1))
q1
#> <quosure>
#> expr: ^x
#> env:  0x556a6af560a8

q2 <- new_quosure(expr(x + !!q1), env(x = 10))
q2
#> <quosure>
#> expr: ^x + (^x)
#> env:  0x556a6b0cc7f0

q3 <- new_quosure(expr(x + !!q2), env(x = 100))
q3
```

```
#> <quosure>
#> expr: ^x + (^x + (^x))
#> env:  0x556a6b604980
```

2. Escriba una función `enenv()` que capture el entorno asociado con un argumento. (Sugerencia: esto solo debería requerir dos llamadas de función).

20.4. Máscaras de datos

En esta sección, aprenderá sobre la **máscara de datos**, un marco de datos donde el código evaluado buscará primero definiciones de variables. La máscara de datos es la idea clave que impulsa funciones básicas como `with()`, `subset()` y `transform()`, y se usa en todo el tidyverse en paquetes como dplyr y ggplot2.

20.4.1. Lo escencial

La máscara de datos le permite combinar variables de un entorno y un marco de datos en una sola expresión. Usted proporciona la máscara de datos como segundo argumento para `eval_tidy()`:

```
q1 <- new_quosure(expr(x * y), env(x = 100))
df <- data.frame(y = 1:10)

eval_tidy(q1, df)
#>  [1]  100  200  300  400  500  600  700  800  900 1000
```

Este código es un poco difícil de seguir porque hay mucha sintaxis ya que estamos creando cada objeto desde cero. Es más fácil ver lo que está pasando si hacemos un pequeño envoltorio. Llamo a esto `with2()` porque es equivalente a `base::with()`.

```
with2 <- function(data, expr) {
  expr <- enquo(expr)
  eval_tidy(expr, data)
}
```

Ahora podemos reescribir el código anterior de la siguiente manera:

```
x <- 100
with2(df, x * y)
#>  [1]  100  200  300  400  500  600  700  800  900 1000
```

`base::eval()` tiene una funcionalidad similar, aunque no lo llama máscara de datos. En su lugar, puede proporcionar un marco de datos al segundo argumento y un entorno al tercero. Eso da la siguiente implementación de `with()`:

```
with3 <- function(data, expr) {
  expr <- substitute(expr)
  eval(expr, data, caller_env())
}
```

20.4.2. Pronombres

El uso de una máscara de datos introduce ambigüedad. Por ejemplo, en el siguiente código no puede saber si `x` vendrá de la máscara de datos o del entorno, a menos que sepa qué variables se encuentran en `df`.

```
with2(df, x)
```

Eso hace que el código sea más difícil de razonar (porque necesita conocer más contexto), lo que puede introducir errores. Para resolver ese problema, la máscara de datos proporciona dos pronombres: `.data` y `.env`.

- `.data$x` siempre se refiere a `x` en la máscara de datos.
- `.env$x` siempre se refiere a `x` en el entorno.

```
x <- 1
df <- data.frame(x = 2)

with2(df, .data$x)
#> [1] 2
with2(df, .env$x)
#> [1] 1
```

También puede crear subconjuntos de `.data` y `.env` usando `[[`, p. `.data[["x"]]`. De lo contrario, los pronombres son objetos especiales y no debe esperar que se comporten como marcos de datos o entornos. En particular, arrojan un error si no se encuentra el objeto:

```
with2(df, .data$y)
#> Error in `.data$y`:
#> ! Column `y` not found in `.data`.
```

20.4.3. Aplicar: `subset()`

Exploraremos la evaluación ordenada en el contexto de `base::subset()`, porque es una función simple pero poderosa que facilita un desafío común de manipulación de datos. Si no lo ha usado antes, `subset()`, como `dplyr::filter()`, proporciona una manera conveniente de seleccionar filas de un marco de datos. Le das algunos datos, junto con una expresión que se evalúa en el contexto de esos datos. Esto reduce considerablemente la cantidad de veces que necesita escribir el nombre del marco de datos:

```
sample_df <- data.frame(a = 1:5, b = 5:1, c = c(5, 3, 1, 4, 1))

# Abreviatura para sample_df[sample_df$a >= 4, ]
subset(sample_df, a >= 4)
#>   a b c
#> 4 4 2 4
#> 5 5 1 1

# Abreviatura para sample_df[sample_df$b == sample_df$c, ]
subset(sample_df, b == c)
#>   a b c
#> 1 1 5 5
#> 5 5 1 1
```

El núcleo de nuestra versión de `subset()`, `subset2()`, es bastante simple. Toma dos argumentos: un marco de datos, `data`, y una expresión, `row`. Evaluamos `row` usando `df` como una máscara de datos, luego usamos los resultados para dividir el marco de datos con `[`. He incluido una verificación muy simple para garantizar que el resultado sea un vector lógico; el código real haría más para crear un error informativo.

```
subset2 <- function(data, rows) {
  rows <- enquo(rows)
  rows_val <- eval_tidy(rows, data)
  stopifnot(is.logical(rows_val))

  data[rows_val, , drop = FALSE]
}

subset2(sample_df, b == c)
#>   a b c
#> 1 1 5 5
#> 5 5 1 1
```

20.4.4. Aplicar: transform

Una situación más complicada es `base::transform()`, que te permite agregar nuevas variables a un marco de datos, evaluando sus expresiones en el contexto de las variables existentes:

```
df <- data.frame(x = c(2, 3, 1), y = runif(3))
transform(df, x = -x, y2 = 2 * y)
#>    x      y    y2
#> 1 -2 0.0808 0.162
#> 2 -3 0.8343 1.669
#> 3 -1 0.6008 1.202
```

De nuevo, nuestro propio `transform2()` requiere poco código. Capturamos el `...` no evaluado con `enquos(...)`, y luego evaluamos cada expresión usando un bucle for. El código real haría más comprobaciones de errores para garantizar que cada entrada tenga un nombre y se evalúe como un vector de la misma longitud que `data`.

```
transform2 <- function(.data, ...) {
  dots <- enquos(...)

  for (i in seq_along(dots)) {
    name <- names(dots)[[i]]
    dot <- dots[[i]]

    .data[[name]] <- eval_tidy(dot, .data)
  }

  .data
}

transform2(df, x2 = x * 2, y = -y)
```

```
#>   x       y x2
#> 1 2 -0.0808  4
#> 2 3 -0.8343  6
#> 3 1 -0.6008  2
```

NB: Llamé al primer argumento `.data` para evitar problemas si los usuarios intentaran crear una variable llamada `data`. Todavía tendrán problemas si intentan crear una variable llamada `.data`, pero esto es mucho menos probable. Este es el mismo razonamiento que lleva a los argumentos `.x` y `.f` a `map()` (Section 9.2.4).

20.4.5. Aplicar: `select()`

Una máscara de datos suele ser un marco de datos, pero a veces es útil proporcionar una lista llena de contenidos más exóticos. Básicamente, así es como funciona el argumento `select` en `base::subset()`. Te permite referirte a las variables como si fueran números:

```
df <- data.frame(a = 1, b = 2, c = 3, d = 4, e = 5)
subset(df, select = b:d)
#>   b c d
#> 1 2 3 4
```

La idea clave es crear una lista con nombre donde cada componente proporcione la posición de la variable correspondiente:

```
vars <- as.list(set_names(seq_along(df), names(df)))
str(vars)
#> List of 5
#>  $ a: int 1
#>  $ b: int 2
#>  $ c: int 3
```

```
#>  $ d: int 4
#>  $ e: int 5
```

Luego, la implementación es nuevamente solo unas pocas líneas de código:

```
select2 <- function(data, ...) {
  dots <- enquos(...)

  vars <- as.list(set_names(seq_along(data), names(data)))
  cols <- unlist(map(dots, eval_tidy, vars))

  data[, cols, drop = FALSE]
}
select2(df, b:d)
#>   b c d
#> 1 2 3 4
```

`dplyr::select()` toma esta idea y la ejecuta, proporcionando una serie de ayudantes que le permiten seleccionar variables en función de sus nombres (por ejemplo, `starts_with("x")` o `ends_with("_a")`).

20.4.6. Ejercicios

1. ¿Por qué usé un bucle for en `transform2()` en lugar de `map()`? Considere `transform2(df, x = x * 2, x = x * 2)`.

2. Aquí hay una implementación alternativa de `subset2()`:

   ```
   subset3 <- function(data, rows) {
     rows <- enquo(rows)
     eval_tidy(expr(data[!!rows, , drop = FALSE]), data = data)
   }
   ```

```
df <- data.frame(x = 1:3)
subset3(df, x == 1)
```

Compara y contrasta `subset3()` con `subset2()`. ¿Cuáles son sus ventajas y desventajas?

3. La siguiente función implementa los conceptos básicos de `dplyr::arrange()`. Anote cada línea con un comentario que explique lo que hace. ¿Puede explicar por qué `!!.na.last` es estrictamente correcto, pero es poco probable que omitir `!!` cause problemas?

```
arrange2 <- function(.df, ..., .na.last = TRUE) {
  args <- enquos(...)

  order_call <- expr(order(!!!args, na.last = !!.na.last))

  ord <- eval_tidy(order_call, .df)
  stopifnot(length(ord) == nrow(.df))

  .df[ord, , drop = FALSE]
}
```

20.5. Usando una evaluación ordenada

Si bien es importante comprender cómo funciona `eval_tidy()`, la mayoría de las veces no lo llamará directamente. En su lugar, normalmente lo usará indirectamente llamando a una función que usa `eval_tidy()`. Esta sección brindará algunos ejemplos prácticos de funciones de envoltura que utilizan una evaluación ordenada.

20.5.1. Citar y remover cita

Imagina que hemos escrito una función que vuelve a muestrear un conjunto de datos:

```
resample <- function(df, n) {
  idx <- sample(nrow(df), n, replace = TRUE)
  df[idx, , drop = FALSE]
}
```

Queremos crear una nueva función que nos permita volver a muestrear y crear subconjuntos en un solo paso. Nuestro enfoque ingenuo no funciona:

```
subsample <- function(df, cond, n = nrow(df)) {
  df <- subset2(df, cond)
  resample(df, n)
}

df <- data.frame(x = c(1, 1, 1, 2, 2), y = 1:5)
subsample(df, x == 1)
#> Error in eval(expr, envir, enclos): object 'x' not found
```

`subsample()` no cita ningún argumento, por lo que `cond` se evalúa normalmente (no en una máscara de datos), y obtenemos un error cuando intenta encontrar un enlace para `x`. Para solucionar este problema, necesitamos citar `cond`, y luego quitarlo cuando lo pasemos a `subset2()`:

```
subsample <- function(df, cond, n = nrow(df)) {
  cond <- enquo(cond)

  df <- subset2(df, !!cond)
  resample(df, n)
```

```
}

subsample(df, x == 1)
#>   x y
#> 3 1 3
#> 1 1 1
#> 2 1 2
```

Este es un patrón muy común; cada vez que llame a una función de cotización con argumentos del usuario, debe citarlos y luego quitarlos.

20.5.2. Manejo de la ambigüedad

En el caso anterior, necesitábamos pensar en una evaluación ordenada debido a la cuasicita. También debemos pensar en una evaluación ordenada, incluso cuando el contenedor no necesita citar ningún argumento. Tome este envoltorio alrededor de `subset2()`:

```
threshold_x <- function(df, val) {
  subset2(df, x >= val)
}
```

Esta función puede devolver silenciosamente un resultado incorrecto en dos situaciones:

- Cuando `x` existe en el entorno de llamada, pero no en `df`:

  ```
  x <- 10
  no_x <- data.frame(y = 1:3)
  threshold_x(no_x, 2)
  #>   y
  #> 1 1
  ```

```
#> 2 2
#> 3 3
```

- Cuando `val` existe en `df`:

```
has_val <- data.frame(x = 1:3, val = 9:11)
threshold_x(has_val, 2)
#> [1] x   val
#> <0 rows> (or 0-length row.names)
```

Estos modos de falla surgen porque la evaluación ordenada es ambigua: cada variable se puede encontrar en **ya sea** la máscara de datos **o** el entorno. Para que esta función sea segura, necesitamos eliminar la ambigüedad usando los pronombres `.data` y `.env`:

```
threshold_x <- function(df, val) {
  subset2(df, .data$x >= .env$val)
}

x <- 10
threshold_x(no_x, 2)
#> Error in `.data$x`:
#> ! Column `x` not found in `.data`.
threshold_x(has_val, 2)
#>   x val
#> 2 2  10
#> 3 3  11
```

Generalmente, cada vez que usa el pronombre `.env`, puede usar la eliminación de comillas en su lugar:

```
threshold_x <- function(df, val) {
  subset2(df, .data$x >= !!val)
}
```

Hay diferencias sutiles en cuando se evalúa `val`. Si elimina las comillas, `val` será evaluado antes por `enquo()`; si usa un pronombre, `val` será evaluado perezosamente por `eval_tidy()`. Estas diferencias generalmente no son importantes, así que elija la forma que se vea más natural.

20.5.3. Citas y ambigüedad

Para finalizar nuestra discusión, consideremos el caso en el que tenemos tanto citas como ambigüedad potencial. Generalizaré `threshold_x()` ligeramente para que el usuario pueda elegir la variable utilizada para el umbral. Aquí usé `.data[[var]]` porque hace que el código sea un poco más simple; en los ejercicios tendrá la oportunidad de explorar cómo podría usar `$` en su lugar.

```
threshold_var <- function(df, var, val) {
  var <- as_string(ensym(var))
  subset2(df, .data[[var]] >= !!val)
}

df <- data.frame(x = 1:10)
threshold_var(df, x, 8)
#>     x
#> 8   8
#> 9   9
#> 10 10
```

No siempre es responsabilidad del autor de la función evitar la ambigüedad. Imagine que generalizamos aún más para permitir el umbral basado en cualquier expresión:

```
threshold_expr <- function(df, expr, val) {
  expr <- enquo(expr)
```

```
  subset2(df, !!expr >= !!val)
}
```

No es posible evaluar `expr` solo en la máscara de datos, porque la máscara de datos no incluye ninguna función como `+` o `==`. Aquí, es responsabilidad del usuario evitar ambigüedades. Como regla general, como autor de una función, es su responsabilidad evitar la ambigüedad con cualquier expresión que cree; es responsabilidad del usuario evitar la ambigüedad en las expresiones que crea.

20.5.4. Ejercicios

1. He incluido una implementación alternativa de `threshold_var()` a continuación. ¿Qué lo hace diferente al enfoque que usé anteriormente? ¿Qué lo hace más difícil?

```
threshold_var <- function(df, var, val) {
  var <- ensym(var)
  subset2(df, `$`(.data, !!var) >= !!val)
}
```

20.6. Evaluación base

Ahora que comprende la evaluación ordenada, es hora de volver a los enfoques alternativos tomados por la base R. Aquí exploraré los dos usos más comunes en la base R:

- `substitute()` y evaluación en el entorno de la persona que llama, tal como lo utiliza `subset()`. Usaré esta técnica para demostrar por qué esta técnica no es fácil de programar, como se advierte en la documentación `subset()`.

- `match.call()`, manipulación de llamadas y evaluación en el entorno de la persona que llama, como lo usan `write.csv()` y `lm()`. Usaré esta técnica para demostrar cómo la cuasicita y la evaluación (regular) pueden ayudarlo a escribir envolturas alrededor de tales funciones.

Estos dos enfoques son formas comunes de evaluación no estándar (NSE).

20.6.1. `substitute()`

La forma más común de NSE en base R es `substitute()` + `eval()`. El siguiente código muestra cómo puedes escribir el núcleo de `subset()` en este estilo usando `substitute()` y `eval()` en lugar de `enquo()` y `eval_tidy()`. Repito el código introducido en la Section 20.4.3 para que puedas comparar fácilmente. La principal diferencia es el entorno de evaluación: en `subset_base()`, el argumento se evalúa en el entorno de la persona que llama, mientras que en `subset_tidy()`, se evalúa en el entorno en el que se definió.

```
subset_base <- function(data, rows) {
  rows <- substitute(rows)
  rows_val <- eval(rows, data, caller_env())
  stopifnot(is.logical(rows_val))

  data[rows_val, , drop = FALSE]
}

subset_tidy <- function(data, rows) {
  rows <- enquo(rows)
  rows_val <- eval_tidy(rows, data)
  stopifnot(is.logical(rows_val))
```

```
  data[rows_val, , drop = FALSE]
}
```

20.6.1.1. Programación con `subset()`

La documentación de `subset()` incluye la siguiente advertencia:

> Esta es una función de conveniencia diseñada para uso interactivo. Para la programación, es mejor usar las funciones estándar de creación de subconjuntos como `[` y, en particular, la evaluación no estándar del argumento `subconjunto` puede tener consecuencias imprevistas.

Hay tres problemas principales:

- `base::subset()` siempre evalúa `rows` en el entorno de llamada, pero si se ha utilizado `...`, es posible que la expresión deba evaluarse en otro lugar:

  ```
  f1 <- function(df, ...) {
    xval <- 3
    subset_base(df, ...)
  }

  my_df <- data.frame(x = 1:3, y = 3:1)
  xval <- 1
  f1(my_df, x == xval)
  #>   x y
  #> 3 3 1
  ```

  Esto puede parecer una preocupación esotérica, pero significa que `subset_base()` no puede funcionar de manera confiable con funciones como `map()` o `lapply()`:

```
local({
  zzz <- 2
  dfs <- list(data.frame(x = 1:3), data.frame(x = 4:6))
  lapply(dfs, subset_base, x == zzz)
})
#> Error in eval(rows, data, caller_env()): object 'zzz' not found
```

- Llamar a `subset()` desde otra función requiere cierto cuidado: debe usar `substitute()` para capturar una llamada a la expresión completa de `subset()` y luego evaluar. Creo que este código es difícil de entender porque `substitute()` no usa un marcador sintáctico para quitar las comillas. Aquí imprimo la llamada generada para que sea un poco más fácil ver lo que está pasando.

```
f2 <- function(df1, expr) {
  call <- substitute(subset_base(df1, expr))
  expr_print(call)
  eval(call, caller_env())
}

my_df <- data.frame(x = 1:3, y = 3:1)
f2(my_df, x == 1)
#> subset_base(my_df, x == 1)
#>   x y
#> 1 1 3
```

- `eval()` no proporciona ningún pronombre, por lo que no hay forma de exigir que parte de la expresión provenga de los datos. Por lo que puedo decir, no hay forma de hacer que la siguiente función sea segura, excepto comprobando manualmente la presencia de la variable `z` en `df`.

```
f3 <- function(df) {
  call <- substitute(subset_base(df, z > 0))
  expr_print(call)
```

```
  eval(call, caller_env())
}

my_df <- data.frame(x = 1:3, y = 3:1)
z <- -1
f3(my_df)
#> subset_base(my_df, z > 0)
#> [1] x y
#> <0 rows> (or 0-length row.names)
```

20.6.1.2. ¿Qué pasa con `[`?

Dado que la evaluación ordenada es bastante compleja, ¿por qué no usar simplemente `[` como recomienda `?subset`? Principalmente, me parece poco atractivo tener funciones que solo se pueden usar de forma interactiva y nunca dentro de otra función.

Además, incluso la función simple `subset()` proporciona dos características útiles en comparación con `[`:

- Establece `drop = FALSE` de forma predeterminada, por lo que se garantiza que devolverá un marco de datos.
- Elimina las filas donde la condición se evalúa como `NA`.

Eso significa que `subset(df, x == y)` no es equivalente a `df[x == y,]` como cabría esperar. En cambio, es equivalente a `df[x == y & !is.na(x == y), , drop = FALSE]`: ¡eso es mucho más tipeo! Las alternativas de la vida real a `subset()`, como `dplyr::filter()`, hacen aún más. Por ejemplo, `dplyr::filter()` puede traducir expresiones R a SQL para que puedan ejecutarse en una base de datos. Esto hace que programar con `filter()` sea relativamente más importante.

20.6.2. `match.call()`

Otra forma común de NSE es capturar la llamada completa con `match.call()`, modificarla y evaluar el resultado. `match.call()` es similar a `substitute()`, pero en lugar de capturar un solo argumento, captura la llamada completa. No tiene un equivalente en rlang.

```
g <- function(x, y, z) {
  match.call()
}
g(1, 2, z = 3)
#> g(x = 1, y = 2, z = 3)
```

Un usuario destacado de `match.call()` es `write.csv()`, que básicamente funciona transformando la llamada en una llamada a `write.table()` con los argumentos adecuados establecidos. El siguiente código muestra el corazón de `write.csv()`:

```
write.csv <- function(...) {
  call <- match.call(write.table, expand.dots = TRUE)

  call[[1]] <- quote(write.table)
  call$sep <- ","
  call$dec <- "."

  eval(call, parent.frame())
}
```

No creo que esta técnica sea una buena idea porque puedes lograr el mismo resultado sin NSE:

```
write.csv <- function(...) {
  write.table(..., sep = ",", dec = ".")
}
```

Sin embargo, es importante comprender esta técnica porque se usa comúnmente en el modelado de funciones. Estas funciones también imprimen de forma destacada la llamada capturada, lo que plantea algunos desafíos especiales, como verá a continuación.

20.6.2.1. Envolviendo funciones de modelado

Para comenzar, considere el envoltorio más simple posible alrededor de `lm()`:

```
lm2 <- function(formula, data) {
  lm(formula, data)
}
```

Este contenedor funciona, pero no es óptimo porque `lm()` captura su llamada y la muestra al imprimir.

```
lm2(mpg ~ disp, mtcars)
#>
#> Call:
#> lm(formula = formula, data = data)
#>
#> Coefficients:
#> (Intercept)         disp
#>     29.5999      -0.0412
```

Arreglar esto es importante porque esta llamada es la forma principal en que ve la especificación del modelo al imprimir el modelo. Para superar

este problema, necesitamos capturar los argumentos, crear la llamada a `lm()` sin comillas y luego evaluar esa llamada. Para que sea más fácil ver lo que está pasando, también imprimiré la expresión que generamos. Esto será más útil a medida que las llamadas se vuelvan más complicadas.

```
lm3 <- function(formula, data, env = caller_env()) {
  formula <- enexpr(formula)
  data <- enexpr(data)

  lm_call <- expr(lm(!!formula, data = !!data))
  expr_print(lm_call)
  eval(lm_call, env)
}

lm3(mpg ~ disp, mtcars)
#> lm(mpg ~ disp, data = mtcars)
#>
#> Call:
#> lm(formula = mpg ~ disp, data = mtcars)
#>
#> Coefficients:
#> (Intercept)         disp
#>     29.5999      -0.0412
```

Hay tres piezas que usará cada vez que envuelva una función NSE base de esta manera:

- Captura los argumentos no evaluados usando `enexpr()`, y captura el entorno de la persona que llama usando `caller_env()`.
- Generas una nueva expresión usando `expr()` y quitando las comillas.
- Evalúa esa expresión en el entorno de la persona que llama. Debe aceptar que la función no funcionará correctamente si los argumentos

no están definidos en el entorno de la persona que llama. Proporcionar el argumento `env` al menos proporciona un gancho que los expertos pueden usar si el entorno predeterminado no es correcto.

El uso de `enexpr()` tiene un buen efecto secundario: podemos usar la eliminación de comillas para generar fórmulas dinámicamente:

```
resp <- expr(mpg)
disp1 <- expr(vs)
disp2 <- expr(wt)
lm3(!!resp ~ !!disp1 + !!disp2, mtcars)
#> lm(mpg ~ vs + wt, data = mtcars)
#>
#> Call:
#> lm(formula = mpg ~ vs + wt, data = mtcars)
#>
#> Coefficients:
#> (Intercept)           vs           wt
#>       33.00         3.15        -4.44
```

20.6.2.2. Entorno de evaluación

¿Qué sucede si desea mezclar objetos proporcionados por el usuario con objetos que crea en la función? Por ejemplo, imagina que quieres hacer una versión de remuestreo automático de `lm()`. Podrías escribirlo así:

```
resample_lm0 <- function(formula, data, env = caller_env()) {
  formula <- enexpr(formula)
  resample_data <- resample(data, n = nrow(data))

  lm_call <- expr(lm(!!formula, data = resample_data))
  expr_print(lm_call)
  eval(lm_call, env)
```

```
}

df <- data.frame(x = 1:10, y = 5 + 3 * (1:10) + round(rnorm(10), 2))
resample_lm0(y ~ x, data = df)
#> lm(y ~ x, data = resample_data)
#> Error in eval(mf, parent.frame()): object 'resample_data' not found
```

¿Por qué no funciona este código? Estamos evaluando `lm_call` en el entorno de la persona que llama, pero `resample_data` existe en el entorno de ejecución. En cambio, podríamos evaluar en el entorno de ejecución de `resample_lm0()`, pero no hay garantía de que `formula` pueda evaluarse en ese entorno.

Hay dos formas básicas de superar este desafío:

1. Elimine las comillas del marco de datos en la llamada. Esto significa que no tiene que ocurrir ninguna búsqueda, pero tiene todos los problemas de las expresiones en línea (Section 19.4.7). Para las funciones de modelado, esto significa que la llamada capturada no es óptima:

```
resample_lm1 <- function(formula, data, env = caller_env()) {
  formula <- enexpr(formula)
  resample_data <- resample(data, n = nrow(data))

  lm_call <- expr(lm(!!formula, data = !!resample_data))
  expr_print(lm_call)
  eval(lm_call, env)
}
resample_lm1(y ~ x, data = df)$call
#> lm(y ~ x, data = <data.frame>)
#> lm(formula = y ~ x, data = list(x = c(3L, 7L, 4L, 4L,
#> 2L, 7L, 2L, 1L, 8L, 9L), y = c(13.21, 27.04, 18.63,
#> 18.63, 10.99, 27.04, 10.99, 7.83, 28.14, 32.72)))
```

2. Alternativamente, puede crear un nuevo entorno que herede de la persona que llama y vincular las variables que ha creado dentro de la función a ese entorno.

```
resample_lm2 <- function(formula, data, env = caller_env()) {
  formula <- enexpr(formula)
  resample_data <- resample(data, n = nrow(data))

  lm_env <- env(env, resample_data = resample_data)
  lm_call <- expr(lm(!!formula, data = resample_data))
  expr_print(lm_call)
  eval(lm_call, lm_env)
}
resample_lm2(y ~ x, data = df)
#> lm(y ~ x, data = resample_data)
#>
#> Call:
#> lm(formula = y ~ x, data = resample_data)
#>
#> Coefficients:
#> (Intercept)            x
#>        4.42         3.11
```

Esto es más trabajo, pero da la especificación más limpia.

20.6.3. Ejercicios

1. ¿Por qué falla esta función?

```
lm3a <- function(formula, data) {
  formula <- enexpr(formula)

  lm_call <- expr(lm(!!formula, data = data))
  eval(lm_call, caller_env())
```

```
}
lm3a(mpg ~ disp, mtcars)$call
#> Error en as.data.frame.default(data, optional = TRUE):
#> no puede obligar a la clase '"function"' a un data.frame
```

2. Cuando se crea un modelo, normalmente la respuesta y los datos son relativamente constantes mientras experimenta rápidamente con diferentes predictores. Escriba un pequeño contenedor que le permita reducir la duplicación en el código a continuación.

```
lm(mpg ~ disp, data = mtcars)
lm(mpg ~ I(1 / disp), data = mtcars)
lm(mpg ~ disp * cyl, data = mtcars)
```

3. Otra forma de escribir `resample_lm()` sería incluir la expresión de remuestreo (`data[sample(nrow(data), replace = TRUE), , drop = FALSE]`) en el argumento de datos. Implemente ese enfoque. ¿Cuáles son las ventajas? ¿Cuales son las desventajas?

21. Traducir código R

21.1. Introducción

La combinación de entornos de primera clase, alcance léxico y metaprogramación nos brinda un poderoso conjunto de herramientas para traducir código R a otros lenguajes. Un ejemplo completo de esta idea es dbplyr, que impulsa los backends de la base de datos para dplyr, lo que le permite expresar la manipulación de datos en R y traducirlos automáticamente a SQL. Puedes ver la idea clave en `translate_sql()` que toma el código R y devuelve el SQL equivalente:

```
library(dbplyr)

con <- simulate_dbi()

translate_sql(x ^ 2, con = con)
#> <SQL> POWER(`x`, 2.0)
translate_sql(x < 5 & !is.na(x), con = con)
#> <SQL> `x` < 5.0 AND NOT((`x` IS NULL))
translate_sql(!first %in% c("John", "Roger", "Robert"), con = con)
#> <SQL> NOT(`first` IN ('John', 'Roger', 'Robert'))
translate_sql(select == 7, con = con)
#> <SQL> `select` = 7.0
```

Traducir R a SQL es complejo debido a las muchas idiosincrasias de los dialectos de SQL, por lo que aquí desarrollaré dos lenguajes específicos de

dominio (DSL) simples pero útiles: uno para generar HTML y el otro para generar ecuaciones matemáticas en LaTeX.

Si está interesado en obtener más información sobre los idiomas específicos del dominio en general, le recomiendo *Idiomas específicos del dominio* (Fowler 2010). Analiza muchas opciones para crear un DSL y proporciona muchos ejemplos de diferentes idiomas.

Estructura

- La Section 21.2 crea un DSL para generar HTML, usando quasiquotation y purrr para generar una función para cada etiqueta HTML, luego ordena la evaluación para acceder fácilmente a ellos.

- La Section 21.3) transforma matemáticamente el código R en su equivalente LaTeX usando una combinación de evaluación ordenada y caminata de expresión.

Requisitos previos

Este capítulo reúne muchas técnicas discutidas en otras partes del libro. En particular, deberá comprender los entornos, las expresiones, la evaluación ordenada y un poco de programación funcional y S3. Usaremos rlang para herramientas de metaprogramación y purrr para programación funcional.

```
library(rlang)
library(purrr)
```

21.2. HTML

HTML (lenguaje de marcado de hipertexto) subyace en la mayor parte de la web. Es un caso especial de SGML (Lenguaje de marcado generalizado estándar), y es similar pero no idéntico a XML (Lenguaje de marcado extensible). HTML se ve así:

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

Incluso si nunca antes ha mirado HTML, aún puede ver que el componente clave de su estructura de codificación son las etiquetas, que se ven como `<tag></tag>` o `<tag/>`. Las etiquetas se pueden anidar dentro de otras etiquetas y entremezclarse con el texto. Hay más de 100 etiquetas HTML, pero en este capítulo nos centraremos en unas pocas:

- `<body>` es la etiqueta de nivel superior que contiene todo el contenido.
- `<h1>` define un encabezado de nivel superior.
- `<p>` define un párrafo.
- `<b>` texto envalentonado.
- `<img>` incrusta una imagen.

Las etiquetas pueden tener **atributos** con nombre que se parecen a `<tag nombre1='valor1' nombre2='valor2'></tag>`. Dos de los atributos más importantes son `id` y `class`, que se utilizan junto con CSS (hojas de estilo en cascada) para controlar la apariencia visual de la página.

Etiquetas anuladas, como `<img>`, no tienen hijos y se escriben `<img />`, no `<img></img>`. Dado que no tienen contenido, los atributos son más

importantes, y `img` tiene tres que se usan con casi todas las imágenes: `src` (donde vive la imagen), `width` y `height`.

Debido a que `<` y `>` tienen significados especiales en HTML, no puede escribirlos directamente. En su lugar, debe usar los **escapes** de HTML: `>` y `<`. Y dado que esos escapes usan `&`, si quieres un ampersand literal, tienes que escapar como `&`.

21.2.1. Objetivo

Nuestro objetivo es facilitar la generación de HTML desde R. Para dar un ejemplo concreto, queremos generar el siguiente HTML:

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

Usando el siguiente código que coincida lo más posible con la estructura del HTML:

```
with_html(
  body(
    h1("Un encabezado", id = "first"),
    p("Un poco de textp &", b("un poco de texto en negrita.")),
    img(src = "myimg.png", width = 100, height = 100)
  )
)
```

Este DSL tiene las siguientes tres propiedades:

- El anidamiento de llamadas a funciones coincide con el anidamiento de etiquetas.
- Los argumentos sin nombre se convierten en el contenido de la etiqueta y los argumentos con nombre se convierten en sus atributos.
- `&` y otros caracteres especiales se escapan automáticamente.

21.2.2. Escapar

Escapar es tan fundamental para la traducción que será nuestro primer tema. Hay dos desafíos relacionados:

- En la entrada del usuario, necesitamos escapar automáticamente `&`, `<` y `>`.
- Al mismo tiempo, debemos asegurarnos de que `&`, `<` y `>` que generamos no tengan doble escape (es decir, que no generemos accidentalmente `&amp;`, `&lt ;` y `&gt;`).

La forma más sencilla de hacer esto es crear una clase S3 (Section 13.3) que distinga entre texto normal (que necesita escape) y HTML (que no).

```
html <- function(x) structure(x, class = "advr_html")

print.advr_html <- function(x, ...) {
  out <- paste0("<HTML> ", x)
  cat(paste(strwrap(out), collapse = "\n"), "\n", sep = "")
}
```

Luego escribimos un escape genérico. Tiene dos métodos importantes:

- `escape.character()` toma un vector de caracteres regular y devuelve un vector HTML con caracteres especiales (`&`, `<`, `>`) escapados.

- `escape.advr_html()` deja solo el HTML escapado.

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

escape.character <- function(x) {
  x <- gsub("&", "&amp;", x)
  x <- gsub("<", "&lt;", x)
  x <- gsub(">", "&gt;", x)

  html(x)
}

escape.advr_html <- function(x) x
```

Ahora comprobamos que funciona

```
escape("This is some text.")
#> <HTML> This is some text.
escape("x > 1 & y < 2")
#> <HTML> x &gt; 1 &amp; y &lt; 2

# Doble escape no es un problema
escape(escape("This is some text. 1 > 2"))
#> <HTML> This is some text. 1 &gt; 2

# Y el texto que sabemos que es HTML no se escapa.
escape(html("<hr />"))
#> <HTML> <hr />
```

Convenientemente, esto también permite que un usuario opte por nuestro escape si sabe que el contenido ya está escapado.

21.2.3. Funciones básicas de etiquetas

A continuación, escribiremos una función de una etiqueta a mano, luego descubriremos cómo generalizarla para que podamos generar una función para cada etiqueta con código.

Comencemos con `<p>`. Las etiquetas HTML pueden tener tanto atributos (por ejemplo, id o clase) como elementos secundarios (como `<b>` o `<i>`). Necesitamos alguna forma de separarlos en la llamada a la función. Dado que los atributos tienen nombre y los hijos no, parece natural usar argumentos con nombre y sin nombre para ellos, respectivamente. Por ejemplo, una llamada a `p()` podría verse así:

```
p("Texo. ", b(i("some bold italic text")), class = "mypara")
```

Podríamos enumerar todos los atributos posibles de la etiqueta `<p>` en la definición de la función, pero eso es difícil porque hay muchos atributos y porque es posible usar [atributos personalizados](http://html5doctor.com/html5- atributos-de-datos-personalizados/). En su lugar, usaremos `...` y separaremos los componentes en función de si tienen nombre o no. Con esto en mente, creamos una función auxiliar que envuelve `rlang::list2()` (Section 19.6) y devuelve los componentes con nombre y sin nombre por separado:

```
dots_partition <- function(...) {
  dots <- list2(...)

 if (is.null(names(dots))) {
  is_named <- rep(FALSE, length(dots))
} else {
  is_named <- names(dots) != ""
}
```

```
  list(
    named = dots[is_named],
    unnamed = dots[!is_named]
  )
}

str(dots_partition(a = 1, 2, b = 3, 4))
#> List of 2
#>  $ named  :List of 2
#>   ..$ a: num 1
#>   ..$ b: num 3
#>  $ unnamed:List of 2
#>   ..$ : num 2
#>   ..$ : num 4
```

Ahora podemos crear nuestra función `p()`. Note que hay una nueva función aquí: `html_attributes()`. Toma una lista con nombre y devuelve la especificación del atributo HTML como una cadena. Es un poco complicado (en parte, porque trata con algunas idiosincrasias de HTML que no he mencionado aquí), pero no es tan importante y no introduce nuevas ideas de programación, así que no lo discutiré en detalle. Puede encontrar la fuente en línea si desea resolverlo usted mismo.

```
source("dsl-html-attributes.r")
p <- function(...) {
  dots <- dots_partition(...)
  attribs <- html_attributes(dots$named)
  children <- map_chr(dots$unnamed, escape)

  html(paste0(
    "<p", attribs, ">",
    paste(children, collapse = ""),
    "</p>"
```

```
  ))
}

p("Some text")
#> <HTML> <p>Some text</p>
p("Some text", id = "myid")
#> <HTML> <p id='myid'>Some text</p>
p("Some text", class = "important", `data-value` = 10)
#> <HTML> <p class='important' data-value='10'>Some text</p>
```

21.2.4. Funciones de etiquetas

Es sencillo adaptar `p()` a otras etiquetas: solo necesitamos reemplazar `"p"` con el nombre de la etiqueta. Una forma elegante de hacerlo es crear una función con `rlang::new_function()` (Section 19.7.4), utilizando la eliminación de comillas y `paste0()` para generar las etiquetas de inicio y finalización.

```
tag <- function(tag) {
  new_function(
    exprs(... = ),
    expr({
      dots <- dots_partition(...)
      attribs <- html_attributes(dots$named)
      children <- map_chr(dots$unnamed, escape)

      html(paste0(
        !!paste0("<", tag), attribs, ">",
        paste(children, collapse = ""),
        !!paste0("</", tag, ">")
      ))
    }),
```

```
    caller_env()
  )
}
tag("b")
#> function (...)
#> {
#>     dots <- dots_partition(...)
#>     attribs <- html_attributes(dots$named)
#>     children <- map_chr(dots$unnamed, escape)
#>     html(paste0("<b", attribs, ">", paste(children, collapse = ""),
#>         "</b>"))
#> }
```

Necesitamos la extraña sintaxis `exprs(... = )` para generar el argumento vacío `...` en la función de etiqueta. Consulte la Section 18.6.2 para obtener más detalles.

Ahora podemos ejecutar nuestro ejemplo anterior:

```
p <- tag("p")
b <- tag("b")
i <- tag("i")
p("Some text. ", b(i("some bold italic text")), class = "mypara")
#> <HTML> <p class='mypara'>Some text. <b><i>some bold italic
#> text</i></b></p>
```

Antes de generar funciones para cada etiqueta HTML posible, necesitamos crear una variante que maneje etiquetas vacías. `void_tag()` es bastante similar a `tag()`, pero arroja un error si hay etiquetas secundarias, como lo capturan los puntos sin nombre. La etiqueta en sí también se ve un poco diferente.

```
void_tag <- function(tag) {
  new_function(
    exprs(... = ),
    expr({
      dots <- dots_partition(...)
      if (length(dots$unnamed) > 0) {
        abort(!!paste0("<", tag, "> must not have unnamed arguments"))
      }
      attribs <- html_attributes(dots$named)

      html(paste0(!!paste0("<", tag), attribs, " />"))
    }),
    caller_env()
  )
}

img <- void_tag("img")
img
#> function (...)
#> {
#>     dots <- dots_partition(...)
#>     if (length(dots$unnamed) > 0) {
#>         abort("<img> must not have unnamed arguments")
#>     }
#>     attribs <- html_attributes(dots$named)
#>     html(paste0("<img", attribs, " />"))
#> }
img(src = "myimage.png", width = 100, height = 100)
#> <HTML> <img src='myimage.png' width='100' height='100' />
```

21.2.5. Procesando todas las etiquetas

A continuación, debemos generar estas funciones para cada etiqueta. Comenzaremos con una lista de todas las etiquetas HTML:

```
tags <- c("a", "abbr", "address", "article", "aside", "audio",
  "b","bdi", "bdo", "blockquote", "body", "button", "canvas",
  "caption","cite", "code", "colgroup", "data", "datalist",
  "dd", "del","details", "dfn", "div", "dl", "dt", "em",
  "eventsource","fieldset", "figcaption", "figure", "footer",
  "form", "h1", "h2", "h3", "h4", "h5", "h6", "head", "header",
  "hgroup", "html", "i","iframe", "ins", "kbd", "label",
  "legend", "li", "mark", "map","menu", "meter", "nav",
  "noscript", "object", "ol", "optgroup", "option", "output",
  "p", "pre", "progress", "q", "ruby", "rp","rt", "s", "samp",
  "script", "section", "select", "small", "span", "strong",
  "style", "sub", "summary", "sup", "table", "tbody", "td",
  "textarea", "tfoot", "th", "thead", "time", "title", "tr",
  "u", "ul", "var", "video"
)

void_tags <- c("area", "base", "br", "col", "command", "embed",
  "hr", "img", "input", "keygen", "link", "meta", "param",
  "source", "track", "wbr"
)
```

Si observa esta lista detenidamente, verá que hay bastantes etiquetas que tienen el mismo nombre que las funciones base de R (`body`, `col`, `q`, `source`, `sub`, `summary`, `tabla`). Esto significa que no queremos que todas las funciones estén disponibles de forma predeterminada, ya sea en el entorno global o en un paquete. En su lugar, los pondremos en una lista (como en la Section 10.5) y luego proporcionaremos una ayuda para que sea más fácil usarlos cuando se desee. Primero, hacemos una lista con nombre que contiene todas las funciones de etiqueta:

```
html_tags <- c(
  tags %>% set_names() %>% map(tag),
  void_tags %>% set_names() %>% map(void_tag)
)
```

Esto nos da una forma explícita (pero detallada) de crear HTML:

```
html_tags$p(
  "Some text. ",
  html_tags$b(html_tags$i("some bold italic text")),
  class = "mypara"
)
#> <HTML> <p class='mypara'>Some text. <b><i>some bold italic
#> text</i></b></p>
```

Entonces podemos terminar nuestro DSL HTML con una función que nos permita evaluar el código en el contexto de esa lista. Aquí abusamos ligeramente de la máscara de datos, pasándole una lista de funciones en lugar de un marco de datos. Este es un truco rápido para mezclar el entorno de ejecución del `código` con las funciones en `html_tags`.

```
with_html <- function(code) {
  code <- enquo(code)
  eval_tidy(code, html_tags)
}
```

Esto nos brinda una API sucinta que nos permite escribir HTML cuando lo necesitamos, pero no abarrota el espacio de nombres cuando no lo necesitamos.

```
with_html(
  body(
    h1("A heading", id = "first"),
    p("Some text &", b("some bold text.")),
    img(src = "myimg.png", width = 100, height = 100)
  )
)
#> <HTML> <body><h1 id='first'>A heading</h1><p>Some text
#> &amp;<b>some bold text.</b></p><img src='myimg.png'
#> width='100' height='100' /></body>
```

Si desea acceder a la función R anulada por una etiqueta HTML con el mismo nombre dentro de `with_html()`, puede usar la especificación `package::function` completa.

21.2.6. Ejercicios

1. Las reglas de escape para las etiquetas `<script>` son diferentes porque contienen JavaScript, no HTML. En lugar de escapar los corchetes angulares o los símbolos de unión, debe escapar `</script>` para que la etiqueta no se cierre demasiado pronto. Por ejemplo, `script("'</script>'")`, no debería generar esto:

   ```
   <script>'</script>'</script>
   ```

   But

   ```
   <script>'<\/script>'</script>
   ```

   Adapte `escape()` para seguir estas reglas cuando un nuevo argumento `script` se establezca en `TRUE`.

2. El uso de `...` para todas las funciones tiene algunas desventajas importantes. No hay validación de entrada y habrá poca información en la documentación o autocompletar sobre cómo se usan en la función. Cree una nueva función que, cuando se le proporcione una lista con nombre de etiquetas y sus nombres de atributos (como se muestra a continuación), cree funciones de etiqueta con argumentos con nombre.

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

   Todas las etiquetas deben obtener los atributos `class` e `id`.

3. Razona sobre el siguiente código que llama `with_html()` haciendo referencia a objetos del entorno. ¿Funcionará o fallará? ¿Por qué? Ejecute el código para verificar sus predicciones.

```
greeting <- "Hello!"
with_html(p(greeting))

p <- function() "p"
address <- "123 anywhere street"
with_html(p(address))
```

4. Actualmente, el HTML no se ve muy bonito y es difícil ver la estructura. ¿Cómo podrías adaptar `tag()` para sangrar y formatear? (Es posible que deba investigar un poco sobre las etiquetas en bloque y en línea).

21.3. LaTeX

El próximo DSL convertirá las expresiones R en sus equivalentes matemáticos de LaTeX. (Esto es un poco como `?plotmath`, pero para texto en lugar de gráficos.) LaTeX es la lingua franca de los matemáticos y estadísticos: es común usar la notación LaTeX cada vez que desea expresar una ecuación en texto, como en un correo electrónico. Dado que muchos informes se producen con R y LaTeX, podría ser útil poder convertir automáticamente expresiones matemáticas de un idioma a otro.

Debido a que necesitamos convertir funciones y nombres, este DSL matemático será más complicado que el HTML DSL. También necesitaremos crear una conversión predeterminada, para que los símbolos que no conocemos obtengan una conversión estándar. Esto significa que ya no podemos usar solo la evaluación: también necesitamos recorrer el árbol de sintaxis abstracta (AST).

21.3.1. LaTeX matemáticas

Antes de comenzar, veamos rápidamente cómo se expresan las fórmulas en LaTeX. El estándar completo es muy complejo, pero afortunadamente está bien documentado, y los comandos más comunes tienen una estructura bastante simple:

- La mayoría de las ecuaciones matemáticas simples se escriben de la misma manera que las escribirías en R: `x * y`, `z ^ 5`. Los subíndices se escriben usando `_` (por ejemplo, `x_1`).
- Los caracteres especiales comienzan con `\`: `\pi` = π, `\pm` = $\pm$, y así sucesivamente. Hay una gran cantidad de símbolos disponibles en LaTeX: la búsqueda en línea de "símbolos matemáticos de látex" arroja muchas listas. Incluso hay un servicio que buscará el símbolo que dibujes en el navegador.

- Las funciones más complicadas se ven como `\name{arg1}{arg2}`. Por ejemplo, para escribir una fracción usarías `\frac{a}{b}`. Para escribir una raíz cuadrada, usarías `\sqrt{a}`.

- Para agrupar elementos, use `{}`: es decir, `x ^ a + b` versus `x ^ {a + b}`.

- En una buena composición tipográfica matemática, se hace una distinción entre variables y funciones. Pero sin información adicional, LaTeX no sabe si `f(a * b)` representa llamar a la función `f` con la entrada `a * b`, o si es una abreviatura de `f * (a * b)`. Si `f` es una función, puede decirle a LaTeX que la escriba usando una fuente vertical con `\textrm{f}(a * b)`. (El `rm` significa "romano", lo contrario de la cursiva).

21.3.2. Meta

Nuestro objetivo es usar estas reglas para convertir automáticamente una expresión R a su representación LaTeX adecuada. Abordaremos esto en cuatro etapas:

- Convertir símbolos conocidos: `pi` → `\pi`

- Dejar otros símbolos sin cambios: `x` → `x`, `y` → `y`

- Convertir funciones conocidas a sus formas especiales: `sqrt(frac(a, b))` → `\sqrt{\frac{a}{b}}`

- Envuelve funciones desconocidas con `\textrm`: `f(a)` → `\textrm{f}(a)`

Codificaremos esta traducción en la dirección opuesta a lo que hicimos con HTML DSL. Comenzaremos con la infraestructura, porque eso hace que sea fácil experimentar con nuestro DSL, y luego trabajaremos de regreso para generar el resultado deseado.

21.3.3. `to_math()`

Para comenzar, necesitamos una función contenedora que convierta las expresiones R en expresiones matemáticas LaTeX. Esto funcionará como `to_html()` capturando la expresión no evaluada y evaluándola en un entorno especial. Hay dos diferencias principales:

- El entorno de evaluación ya no es constante, ya que tiene que variar según la entrada. Esto es necesario para manejar símbolos y funciones desconocidos.

- Nunca evaluamos en el entorno de argumentos porque estamos traduciendo cada función a una expresión LaTeX. El usuario necesitará usar explícitamente `!!` para evaluar normalmente.

Esto nos da:

```
to_math <- function(x) {
  expr <- enexpr(x)
  out <- eval_bare(expr, latex_env(expr))

  latex(out)
}

latex <- function(x) structure(x, class = "advr_latex")
print.advr_latex <- function(x) {
  cat("<LATEX> ", x, "\n", sep = "")
}
```

A continuación, construiremos `latex_env()`, comenzando de manera simple y haciéndonos progresivamente más complejos.

21.3.4. Símbolos conocidos

Nuestro primer paso es crear un entorno que convierta los símbolos especiales de LaTeX utilizados para los caracteres griegos, por ejemplo, `pi` a `\pi`. Usaremos el truco de la Section 20.4.3 para vincular el símbolo `pi` al valor `"\pi"`.

```
greek <- c(
  "alpha", "theta", "tau", "beta", "vartheta", "pi", "upsilon",
  "gamma", "varpi", "phi", "delta", "kappa", "rho",
  "varphi", "epsilon", "lambda", "varrho", "chi", "varepsilon",
  "mu", "sigma", "psi", "zeta", "nu", "varsigma", "omega", "eta",
  "xi", "Gamma", "Lambda", "Sigma", "Psi", "Delta", "Xi",
  "Upsilon", "Omega", "Theta", "Pi", "Phi"
)
greek_list <- set_names(paste0("\\", greek), greek)
greek_env <- as_environment(greek_list)
```

Entonces podemos comprobarlo:

```
latex_env <- function(expr) {
  greek_env
}

to_math(pi)
#> <LATEX> \pi
to_math(beta)
#> <LATEX> \beta
```

¡Se ve bien hasta ahora!

21.3.5. Símbolos desconocidos

Si un símbolo no es griego, queremos dejarlo como está. Esto es complicado porque no sabemos de antemano qué símbolos se utilizarán y no podemos generarlos todos. En su lugar, usaremos el enfoque descrito en la Section 18.5: recorrer el AST para encontrar todos los símbolos. Esto nos da `all_names_rec()` y el asistente `all_names()`:

```
all_names_rec <- function(x) {
  switch_expr(x,
    constant = character(),
    symbol =   as.character(x),
    call =     flat_map_chr(as.list(x[-1]), all_names)
  )
}

all_names <- function(x) {
  unique(all_names_rec(x))
}

all_names(expr(x + y + f(a, b, c, 10)))
#> [1] "x" "y" "a" "b" "c"
```

Ahora queremos tomar esa lista de símbolos y convertirla en un entorno para que cada símbolo se asigne a su representación de cadena correspondiente (por ejemplo, `eval(quote(x), env)` produce `"x"`). Nuevamente usamos el patrón de convertir un vector de caracteres con nombre en una lista y luego convertir la lista en un entorno.

```
latex_env <- function(expr) {
  names <- all_names(expr)
  symbol_env <- as_environment(set_names(names))
```

```
  symbol_env
}

to_math(x)
#> <LATEX> x
to_math(longvariablename)
#> <LATEX> longvariablename
to_math(pi)
#> <LATEX> pi
```

Esto funciona, pero necesitamos combinarlo con el entorno de símbolos griegos. Dado que queremos dar preferencia al griego sobre los valores predeterminados (por ejemplo, `to_math(pi)` debe dar `"\\pi"`, no `"pi"`), `symbol_env` debe ser el padre de `greek_env`. Para hacer eso, necesitamos hacer una copia de `greek_env` con un nuevo padre. Esto nos da una función que puede convertir símbolos conocidos (griegos) y desconocidos.

```
latex_env <- function(expr) {
  # Símbolos desconocidos
  names <- all_names(expr)
  symbol_env <- as_environment(set_names(names))

  # Símbolos conocidos
  env_clone(greek_env, parent = symbol_env)
}

to_math(x)
#> <LATEX> x
to_math(longvariablename)
#> <LATEX> longvariablename
to_math(pi)
#> <LATEX> \pi
```

21.3.6. Funciones conocidas

A continuación agregaremos funciones a nuestro DSL. Comenzaremos con un par de ayudantes que facilitan la adición de nuevos operadores binarios y unarios. Estas funciones son muy simples: solo ensamblan cadenas.

```
unary_op <- function(left, right) {
  new_function(
    exprs(e1 = ),
    expr(
      paste0(!!left, e1, !!right)
    ),
    caller_env()
  )
}

binary_op <- function(sep) {
  new_function(
    exprs(e1 = , e2 = ),
    expr(
      paste0(e1, !!sep, e2)
    ),
    caller_env()
  )
}

unary_op("\\sqrt{", "}")
#> function (e1)
#> paste0("\\sqrt{", e1, "}")
binary_op("+")
#> function (e1, e2)
#> paste0(e1, "+", e2)
```

Usando estos ayudantes, podemos mapear algunos ejemplos ilustrativos

de conversión de R a LaTeX. Tenga en cuenta que con las reglas de alcance léxico de R ayudándonos, podemos proporcionar fácilmente nuevos significados para funciones estándar como `+`, `-` y `*`, e incluso `(` y `{`.

```
# Operadores binarios
f_env <- child_env(
  .parent = empty_env(),
  `+` = binary_op(" + "),
  `-` = binary_op(" - "),
  `*` = binary_op(" * "),
  `/` = binary_op(" / "),
  `^` = binary_op("^"),
  `[` = binary_op("_"),

  # Agrupamiento
  `{` = unary_op("\\left{ ", " \\right}"),
  `(` = unary_op("\\left( ", " \\right)"),
  paste = paste,

  # Otras funciones matemáticas
  sqrt = unary_op("\\sqrt{", "}"),
  sin =  unary_op("\\sin(", ")"),
  log =  unary_op("\\log(", ")"),
  abs =  unary_op("\\left| ", "\\right| "),
  frac = function(a, b) {
    paste0("\\frac{", a, "}{", b, "}")
  },

  # Etiquetado
  hat =   unary_op("\\hat{", "}"),
  tilde = unary_op("\\tilde{", "}")
)
```

Nuevamente modificamos `latex_env()` para incluir este entorno. Debería

ser el último entorno en el que R busca nombres para que expresiones como `sin(sin)` funcionen.

```
latex_env <- function(expr) {
  # Funciones conocidas
  f_env

  # Símbolos predeterminados
  names <- all_names(expr)
  symbol_env <- as_environment(set_names(names), parent = f_env)

  # Símbolos conocidos
  greek_env <- env_clone(greek_env, parent = symbol_env)

  greek_env
}

to_math(sin(x + pi))
#> <LATEX> \sin(x + \pi)
to_math(log(x[i]^2))
#> <LATEX> \log(x_i^2)
to_math(sin(sin))
#> <LATEX> \sin(sin)
```

21.3.7. Funciones desconocidas

Finalmente, agregaremos un valor predeterminado para las funciones que aún no conocemos. No podemos saber de antemano cuáles serán las funciones desconocidas, por lo que nuevamente recorremos el AST para encontrarlas:

```
all_calls_rec <- function(x) {
  switch_expr(x,
    constant = ,
    symbol =   character(),
    call = {
      fname <- as.character(x[[1]])
      children <- flat_map_chr(as.list(x[-1]), all_calls)
      c(fname, children)
    }
  )
}
all_calls <- function(x) {
  unique(all_calls_rec(x))
}

all_calls(expr(f(g + b, c, d(a))))
#> [1] "f" "+" "d"
```

Necesitamos un cierre que genere las funciones para cada llamada desconocida:

```
unknown_op <- function(op) {
  new_function(
    exprs(... = ),
    expr({
      contents <- paste(..., collapse = ", ")
      paste0(!!paste0("\\mathrm{", op, "}("), contents, ")")
    })
  )
}
unknown_op("foo")
#> function (...)
#> {
```

```
#>     contents <- paste(..., collapse = ", ")
#>     paste0("\\mathrm{foo}(", contents, ")")
#> }
#> <environment: 0x560519f28a98>
```

Y de nuevo la actualizamos `latex_env()`:

```
latex_env <- function(expr) {
  calls <- all_calls(expr)
  call_list <- map(set_names(calls), unknown_op)
  call_env <- as_environment(call_list)

  # Funciones conocidas
  f_env <- env_clone(f_env, call_env)

  # Símbolos predeterminados
  names <- all_names(expr)
  symbol_env <- as_environment(set_names(names), parent = f_env)

  # Símbolos conocidos
  greek_env <- env_clone(greek_env, parent = symbol_env)
  greek_env
}
```

Esto completa nuestros requisitos originales:

```
to_math(sin(pi) + f(a))
#> <LATEX> \sin(\pi) + \mathrm{f}(a)
```

Sin duda, podría llevar esta idea más allá y traducir tipos de expresiones matemáticas, pero no debería necesitar ninguna herramienta de metaprogramación adicional.

21.3.8. Ejercicios

1. Añadir escape. Los símbolos especiales que deben escaparse agregando una barra invertida delante de ellos son `\`, `$` y `%`. Al igual que con HTML, deberá asegurarse de no terminar con doble escape. Por lo tanto, deberá crear una clase S3 pequeña y luego usarla en los operadores de funciones. Eso también le permitirá incrustar LaTeX arbitrario si es necesario.

2. Complete el DSL para admitir todas las funciones que admite `plotmath`.

Part V.

Tecnicas

Introducción

Los últimos cuatro capítulos cubren dos técnicas generales de programación: encontrar y corregir errores, y encontrar y corregir problemas de rendimiento. Las herramientas para medir y mejorar el rendimiento son particularmente importantes porque R no es un lenguaje rápido. Esto no es un accidente: R fue diseñado a propósito para hacer que el análisis de datos interactivo sea más fácil para los humanos, no para hacer que las computadoras sean lo más rápidas posible. Si bien R es lento en comparación con otros lenguajes de programación, para la mayoría de los propósitos, es lo suficientemente rápido. Estos capítulos lo ayudan a manejar los casos en los que R ya no es lo suficientemente rápido, ya sea mejorando el rendimiento de su código R o cambiando a un lenguaje, C++, que está diseñado para el rendimiento.

- Chapter 22 habla de depuración, porque encontrar la causa raíz del error puede ser extremadamente frustrante. Afortunadamente, R tiene algunas herramientas excelentes para la depuración y, cuando se combinan con una estrategia sólida, debería poder encontrar la causa raíz de la mayoría de los problemas de forma rápida y relativamente sencilla.
- Chapter 23 se enfoca en medir el desempeño.
- Chapter 24 luego muestra cómo mejorar el rendimiento.

22. Depuración

22.1. Introducción

¿Qué haces cuando el código R arroja un error inesperado? ¿Qué herramientas tienes para encontrar y solucionar el problema? Este capítulo le enseñará el arte y la ciencia de la depuración, comenzando con una estrategia general y luego siguiendo con herramientas específicas.

Mostraré las herramientas proporcionadas por R y el IDE de RStudio. Recomiendo usar las herramientas de RStudio si es posible, pero también te mostraré los equivalentes que funcionan en todas partes. También puede consultar la documentación oficial de depuración de RStudio que siempre refleja la última versión de RStudio.

NB: No debería necesitar usar estas herramientas al escribir funciones *nuevas*. Si se encuentra usándolos con frecuencia con código nuevo, reconsidere su enfoque. En lugar de tratar de escribir una gran función de una sola vez, trabaje de forma interactiva en piezas pequeñas. Si comienza poco a poco, puede identificar rápidamente por qué algo no funciona y no necesita herramientas de depuración sofisticadas.

Estructura

- La Section 22.2 describe una estrategia general para encontrar y corregir errores.

- La Section 22.3 le presenta la función `traceback()` que le ayuda a localizar exactamente dónde ocurrió un error.
- La Section 22.4 le muestra cómo pausar la ejecución de una función e iniciar un entorno donde puede explorar de forma interactiva lo que está sucediendo.
- La Section 22.5 analiza el desafiante problema de la depuración cuando ejecuta código de forma no interactiva.
- La Section 22.6 analiza un puñado de problemas que no son errores y que ocasionalmente también necesitan depuración.

22.2. Enfoque global

> Encontrar su error es un proceso de confirmación de las muchas cosas que cree que son ciertas, hasta que encuentre una que no lo sea.
>
> —Norm Matloff

Encontrar la causa raíz de un problema siempre es un desafío. La mayoría de los errores son sutiles y difíciles de encontrar porque si fueran obvios, los habrías evitado en primer lugar. Una buena estrategia ayuda. A continuación, describo un proceso de cuatro pasos que he encontrado útil:

1. **Google!**

   Cada vez que vea un mensaje de error, comience a buscarlo en Google. Si tiene suerte, descubrirá que es un error común con una solución conocida. Cuando busque en Google, mejore sus posibilidades de una buena coincidencia eliminando cualquier nombre o valor de variable que sea específico para su problema.

Puede automatizar este proceso con los paquetes errorist (Balamuta 2018a) y searcher (Balamuta 2018b). Consulte sus sitios web para obtener más detalles.

2. **Hazlo repetible**

   Para encontrar la causa raíz de un error, necesitará ejecutar el código muchas veces mientras considera y rechaza las hipótesis. Para que la iteración sea lo más rápida posible, vale la pena hacer una inversión inicial para que el problema sea fácil y rápido de reproducir.

   Comience creando un ejemplo reproducible (Section 1.7). A continuación, haga que el ejemplo sea mínimo eliminando el código y simplificando los datos. Al hacer esto, es posible que descubra entradas que no desencadenan el error. Tome nota de ellos: serán útiles al diagnosticar la causa raíz.

   Si está utilizando pruebas automatizadas, este también es un buen momento para crear un caso de prueba automatizado. Si su cobertura de prueba existente es baja, aproveche la oportunidad para agregar algunas pruebas cercanas para garantizar que se conserve el buen comportamiento existente. Esto reduce las posibilidades de crear un nuevo error.

3. **Averigua dónde está**

   Si tiene suerte, una de las herramientas de la siguiente sección lo ayudará a identificar rápidamente la línea de código que está causando el error. Por lo general, sin embargo, tendrás que pensar un poco más sobre el problema. Es una gran idea adoptar el método científico. Genere hipótesis, diseñe experimentos para probarlas y registre sus resultados. Esto puede parecer mucho trabajo, pero un enfoque sistemático terminará ahorrándole tiempo. A menudo pierdo mucho tiempo confiando en mi intuición para resolver un error ("oh, debe ser un error de uno, así que restaré 1 aquí"), cuando hubiera sido mejor tomar un Acercamiento sistematico.

Si esto falla, es posible que deba pedir ayuda a otra persona. Si ha seguido el paso anterior, tendrá un pequeño ejemplo que es fácil de compartir con otros. Eso hace que sea mucho más fácil para otras personas ver el problema y es más probable que lo ayuden a encontrar una solución.

4. **Arreglarlo y probarlo**

   Una vez que haya encontrado el error, debe descubrir cómo solucionarlo y verificar que la solución realmente funcionó. Una vez más, es muy útil contar con pruebas automatizadas. Esto no solo ayuda a garantizar que realmente haya solucionado el error, sino que también ayuda a garantizar que no haya introducido ningún error nuevo en el proceso. En ausencia de pruebas automatizadas, asegúrese de registrar cuidadosamente la salida correcta y compárela con las entradas que fallaron anteriormente.

22.3. Localización de errores

Una vez que haya hecho que el error sea repetible, el siguiente paso es averiguar de dónde viene. La herramienta más importante para esta parte del proceso es `traceback()`, que le muestra la secuencia de llamadas (también conocida como pila de llamadas, Section 7.5) que conducen al error.

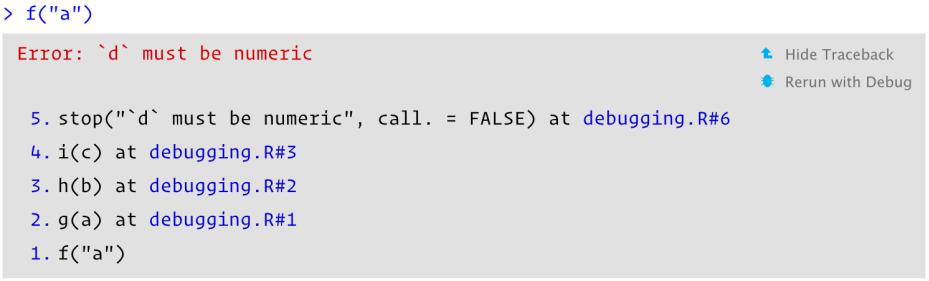
He aquí un ejemplo sencillo: puedes ver que `f()` llama a `g()` llama a `h()` llama a `i()`, que comprueba si su argumento es numérico:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) {
  if (!is.numeric(d)) {
    stop("`d` must be numeric", call. = FALSE)
  }
```

```
  d + 10
}
```

Cuando ejecutamos el código `f("a")` en RStudio vemos:

Aparecen dos opciones a la derecha del mensaje de error: "Mostrar seguimiento" y "Volver a ejecutar con depuración". Si hace clic en "Mostrar seguimiento", verá:



Si no está usando RStudio, puede usar `traceback()` para obtener la misma información (sin un formato bonito):

```
traceback()
#> 5: stop("`d` must be numeric", call. = FALSE) at debugging.R#6
#> 4: i(c) at debugging.R#3
#> 3: h(b) at debugging.R#2
#> 2: g(a) at debugging.R#1
#> 1: f("a")
```

NB: Usted lee la salida `traceback()` de abajo hacia arriba: la llamada inicial es `f()`, que llama `g()`, luego `h()`, luego `i()`, que activa el error. Si está llamando al código que 'fuente () 'd en R, el rastreo también mostrará la ubicación de la función, en la forma `nombredearchivo.r#númerodelínea`. Estos se pueden hacer clic en RStudio y lo llevarán a la línea de código correspondiente en el editor.

22.3.1. Evaluación perezosa

Un inconveniente de `traceback()` es que siempre linealiza el árbol de llamadas, lo que puede ser confuso si hay mucha evaluación perezosa involucrada (Section 7.5.2). Por ejemplo, tome el siguiente ejemplo donde ocurre el error al evaluar el primer argumento de `f()`:

```
j <- function() k()
k <- function() stop("Oops!", call. = FALSE)
f(j())
#> Error: Oops!
```

```
traceback()
#> 7: stop("Oops!") at #1
#> 6: k() at #1
#> 5: j() at debugging.R#1
#> 4: i(c) at debugging.R#3
#> 3: h(b) at debugging.R#2
#> 2: g(a) at debugging.R#1
#> 1: f(j())
```

Puede usar `rlang::with_abort()` y `rlang::last_trace()` para ver el árbol de llamadas. Aquí, creo que hace que sea mucho más fácil ver el origen del problema. Mire la última rama del árbol de llamadas para ver que el error proviene de `j()` llamando a `k()`.

```
rlang::with_abort(f(j()))
#> Error: 'with_abort' is not an exported object from 'namespace:rlang'
rlang::last_trace()
#> Error: Can't show last error because no error was recorded yet
```

NB: `rlang::last_trace()` se ordena de forma opuesta a `traceback()`. Volveremos a ese tema en la Section 22.4.2.4.

22.4. Depurador interactivo

A veces, la ubicación precisa del error es suficiente para permitirle localizarlo y solucionarlo. Sin embargo, con frecuencia necesita más información, y la forma más fácil de obtenerla es con el depurador interactivo que le permite pausar la ejecución de una función y explorar su estado de forma interactiva.

Si está utilizando RStudio, la forma más fácil de ingresar al depurador interactivo es a través de la herramienta "Reejecutar con depuración" de RStudio. Esto vuelve a ejecutar el comando que creó el error, deteniendo la ejecución donde ocurrió el error. De lo contrario, puede insertar una llamada a `browser()` donde desea hacer una pausa y volver a ejecutar la función. Por ejemplo, podríamos insertar una llamada `browser()` en `g()`:

```
g <- function(b) {
  browser()
  h(b)
}
f(10)
```

`browser()` es solo una llamada de función regular, lo que significa que puede ejecutarla condicionalmente envolviéndola en una declaración `if`:

```
g <- function(b) {
  if (b < 0) {
    browser()
  }
  h(b)
}
```

En cualquier caso, terminará en un entorno interactivo *dentro* de la función donde puede ejecutar código R arbitrario para explorar el estado actual. Sabrá cuándo está en el depurador interactivo porque recibe un aviso especial:

```
Browse[1]>
```

En RStudio, verá el código correspondiente en el editor (con la instrucción que se ejecutará a continuación resaltada), los objetos en el entorno actual en el panel Entorno y la pila de llamadas en el panel Rastreo.

22.4.1. Comandos `browser()`

Además de permitirle ejecutar código R regular, `browser()` proporciona algunos comandos especiales. Puede usarlos escribiendo comandos de texto cortos o haciendo clic en un botón en la barra de herramientas de RStudio, Figure 22.1:

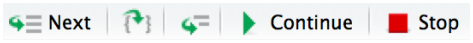


Figure 22.1.: RStudio debugging toolbar

- Siguiente, `n`: ejecuta el siguiente paso en la función. Si tiene una variable llamada `n`, necesitará `print(n)` para mostrar su valor.

- Entrar en,  o `s`: funciona como el siguiente, pero si el siguiente paso es una función, entrará en esa función para que pueda explorarla de forma interactiva.

- Finalizar,  o `f`: finaliza la ejecución del ciclo o función actual.

- Continuar, `c`: sale de la depuración interactiva y continúa con la ejecución normal de la función. Esto es útil si ha solucionado el mal estado y desea comprobar que la función se desarrolla correctamente.

- Detener, `Q`: detiene la depuración, finaliza la función y regresa al espacio de trabajo global. Úselo una vez que haya descubierto dónde está el problema y esté listo para solucionarlo y volver a cargar el código.

Hay otros dos comandos un poco menos útiles que no están disponibles en la barra de herramientas:

- Enter: repite el comando anterior. Encuentro esto demasiado fácil de activar accidentalmente, así que lo apago usando `options(browserNLdisabled = TRUE)`.

- `where`: imprime el seguimiento de la pila de llamadas activas (el equivalente interactivo de `traceback`).

22.4.2. Alternativas

Hay tres alternativas al uso de `browser()`: establecer puntos de interrupción en RStudio, `options(error = recovery)` y `debug()` y otras funciones relacionadas.

22.4.2.1. Puntos de ruptura

En RStudio, puede establecer un punto de interrupción haciendo clic a la izquierda del número de línea o presionando `Shift + F9`. Los puntos de interrupción se comportan de manera similar a `browser()` pero son más fáciles de configurar (un clic en lugar de nueve pulsaciones de teclas), y no corre el riesgo de incluir accidentalmente una declaración `browser()` en su código fuente. Hay dos pequeñas desventajas de los puntos de interrupción:

- Taquí hay algunas situaciones inusuales en las que los puntos de interrupción no funcionarán. Para más detalles lea solución de problemas de puntos de interrupción.
- RStudio actualmente no admite puntos de interrupción condicionales.

22.4.2.2. `recover()`

Otra forma de activar `browser()` es usar `options(error = recover)`. Ahora, cuando reciba un error, obtendrá un mensaje interactivo que muestra el rastreo y le brinda la capacidad de depurar de forma interactiva dentro de cualquiera de los marcos:

```
options(error = recover)
f("x")
#> Error: `d` must be numeric
#>
#> Enter a frame number, or 0 to exit
#>
#> 1: f("x")
#> 2: debugging.R#1: g(a)
#> 3: debugging.R#2: h(b)
#> 4: debugging.R#3: i(c)
```

```
#>
#> Selection:
```

Puede volver al manejo de errores predeterminado con `options(error = NULL)`.

22.4.2.3. `debug()`

Otro enfoque es llamar a una función que inserta la llamada `browser()` por ti:

- `debug()` inserta una declaración del navegador en la primera línea de la función especificada. `undebug()` lo elimina. Alternativamente, puede usar `debugonce()` para navegar solo en la próxima ejecución.
- `utils::setBreakpoint()` funciona de manera similar, pero en lugar de tomar un nombre de función, toma un nombre de archivo y un número de línea y encuentra la función adecuada para usted.

Estas dos funciones son casos especiales de `trace()`, que inserta código arbitrario en cualquier posición de una función existente. `trace()` es ocasionalmente útil cuando estás depurando código para el cual no tienes la fuente. Para eliminar el rastreo de una función, use `untrace()`. Solo puede realizar un seguimiento por función, pero ese seguimiento puede llamar a varias funciones.

22.4.2.4. Pila de llamadas

Desafortunadamente, las pilas de llamadas impresas por `traceback()`, `browser()` & `where`, y `recover()` no son consistentes. La siguiente tabla muestra cómo las tres herramientas muestran las pilas de llamadas de un conjunto anidado simple de llamadas. La numeración es diferente entre

`traceback()` y `where`, y `recover()` muestra las llamadas en el orden opuesto.

`traceback()`	`where`	`recover()`	funciones rlang
`5:`			
`stop("...")`			
`4: i(c)`	`where 1: i(c)`	`1: f()`	`1.  global::f(10)`
`3: h(b)`	`where 2: h(b)`	`2: g(a)`	`2.    global::g(a)`
`2: g(a)`	`where 3: g(a)`	`3: h(b)`	`3.      global::h(b)`
`1: f("a")`	`where 4:`	`4: i("a")`	`4.`
	`f("a")`		`global::i("a")`

RStudio muestra las llamadas en el mismo orden que `traceback()`. Las funciones rlang usan el mismo orden y numeración que `recover()`, pero también usan sangría para reforzar la jerarquía de las llamadas.

22.4.3. Código compilado

También es posible usar un depurador interactivo (gdb o lldb) para código compilado (como C o C++). Desafortunadamente, eso está más allá del alcance de este libro, pero hay algunos recursos que pueden resultarle útiles:

- http://r-pkgs.had.co.nz/src.html#src-debugging
- https://github.com/wch/r-debug/blob/master/debugging-r.md
- http://kevinushey.github.io/blog/2015/04/05/debugging-with-valgrind/
- https://www.jimhester.com/2018/08/22/debugging-rstudio/

22.5. Depuración no interactiva

La depuración es más desafiante cuando no puede ejecutar el código de forma interactiva, generalmente porque es parte de una canalización que se ejecuta automáticamente (posiblemente en otra computadora), o porque el error no ocurre cuando ejecuta el mismo código de forma interactiva. ¡Esto puede ser extremadamente frustrante!

Esta sección le brindará algunas herramientas útiles, pero no olvide la estrategia general en la Section 22.2. Cuando no puede explorar de forma interactiva, es particularmente importante dedicar algún tiempo a hacer que el problema sea lo más pequeño posible para que pueda iterar rápidamente. A veces `callr::r(f, list(1, 2))` puede ser útil; esto llama a `f(1, 2)` en una nueva sesión y puede ayudar a reproducir el problema.

También es posible que desee verificar dos veces estos problemas comunes:

- ¿Es diferente el entorno global? ¿Has cargado diferentes paquetes? ¿Los objetos que quedaron de sesiones anteriores causan diferencias?
- ¿El directorio de trabajo es diferente?
- ¿Es diferente la variable de entorno `PATH`, que determina dónde se encuentran los comandos externos (como `git`)?
- ¿Es diferente la variable de entorno `R_LIBS`, que determina dónde `library()` busca paquetes?

22.5.1. `dump.frames()`

`dump.frames()` es el equivalente a `recover()` para código no interactivo; guarda un archivo `last.dump.rda` en el directorio de trabajo. Más tarde, en una sesión interactiva, puede `load("last.dump.rda"); debugger()` para ingresar a un depurador interactivo con la misma interfaz que

`recover()`. Esto le permite "engañar", depurando de forma interactiva el código que se ejecutó de forma no interactiva.

```
# En proceso por lotes R ----
dump_and_quit <- function() {
  # Guardar información de depuración en un archivo last.dump.rda
  dump.frames(to.file = TRUE)
  # Salir de R con estado de error
  q(status = 1)
}
options(error = dump_and_quit)

# En una sesión interactiva posterior ----
load("last.dump.rda")
debugger()
```

22.5.2. Imprimir depuración

Si `dump.frames()` no ayuda, una buena alternativa es **imprimir la depuración**, donde inserta numerosas instrucciones de impresión para ubicar con precisión el problema y ver los valores de las variables importantes. La depuración de impresión es lenta y primitiva, pero siempre funciona, por lo que es particularmente útil si no puede obtener un buen seguimiento. Comience insertando marcadores de grano grueso y luego hágalos progresivamente más finos a medida que determina exactamente dónde está el problema.

```
f <- function(a) {
  cat("f()\n")
  g(a)
}
g <- function(b) {
  cat("g()\n")
```

```
  cat("b =", b, "\n")
  h(b)
}
h <- function(c) {
  cat("i()\n")
  i(c)
}

f(10)
#> f()
#> g()
#> b = 10
#> i()
#> [1] 20
```

Imprimir la depuración es particularmente útil para el código compilado porque no es raro que el compilador modifique su código hasta el punto de que no pueda descubrir la raíz del problema, incluso cuando se encuentra dentro de un depurador interactivo.

22.5.3. RMarkdown

La depuración del código dentro de los archivos RMarkdown requiere algunas herramientas especiales. Primero, si está tejiendo el archivo usando RStudio, cambie a llamar `rmarkdown::render("path/to/file.Rmd")` en su lugar. Esto ejecuta el código en la sesión actual, lo que facilita la depuración. Si hacer esto hace que el problema desaparezca, deberá descubrir qué hace que los entornos sean diferentes.

Si el problema persiste, deberá usar sus habilidades de depuración interactiva. Independientemente del método que utilice, necesitará un paso adicional: en el controlador de errores, deberá llamar a `sink()`. Esto elimina el sumidero predeterminado que usa knitr para capturar todos los

resultados y garantiza que pueda ver los resultados en la consola. Por ejemplo, para usar `recover()` con RMarkdown, colocaría el siguiente código en su bloque de configuración:

```
options(error = function() {
  sink()
  recover()
})
```

Esto generará una advertencia de "no hay disipador para eliminar" cuando se complete knitr; puede ignorar esta advertencia con seguridad.

Si simplemente quiere un rastreo, la opción más fácil es usar `rlang::trace_back()`, aprovechando la opción `rlang_trace_top_env`. Esto garantiza que solo vea el rastreo de su código, en lugar de todas las funciones llamadas por RMarkdown y knitr.

```
options(rlang_trace_top_env = rlang::current_env())
options(error = function() {
  sink()
  print(rlang::trace_back(bottom = sys.frame(-1)), simplify = "none")
})
```

22.6. Fallos sin error

Hay otras formas de que una función falle además de arrojar un error:

- Una función puede generar una advertencia inesperada. La forma más fácil de rastrear las advertencias es convertirlas en errores con `options(warn = 2)` y usar la pila de llamadas, como `doWithOneRestart()`, `withOneRestart()`, herramientas de depuración regulares. Cuando haga esto, verá algunas llamadas adicionales `withRestarts()` y `.signalSimpleWarning()`. Ignórelos:

son funciones internas que se utilizan para convertir las advertencias en errores.

- Una función puede generar un mensaje inesperado. Puedes usar `rlang::with_abort()` para convertir estos mensajes en errores:

```
f <- function() g()
g <- function() message("Hi!")
f()
#> Hi!

rlang::with_abort(f(), "message")
#> Error: 'with_abort' is not an exported object from 'namespace:rlang'
rlang::last_trace()
#> Error: Can't show last error because no error was recorded yet
```

- Es posible que una función nunca regrese. Esto es particularmente difícil de depurar automáticamente, pero a veces terminar la función y mirar el `traceback()` es informativo. De lo contrario, utilice la depuración de impresión, como en la Section 22.5.2.

- El peor de los escenarios es que su código podría fallar por completo en R, dejándolo sin forma de depurar su código de manera interactiva. Esto indica un error en el código compilado (C o C++).

  Si el error está en su código compilado, deberá seguir los enlaces en la Section 22.4.3 y aprender a usar un depurador de C interactivo (o insertar muchas instrucciones de impresión).

  Si el error está en un paquete o base R, deberá ponerse en contacto con el mantenedor del paquete. En cualquier caso, trabaje para hacer el ejemplo reproducible más pequeño posible (Section 1.7) para ayudar al desarrollador a ayudarlo.

23. Medición de desempeño

23.1. Introducción

> Los programadores pierden enormes cantidades de tiempo pensando o preocupándose por la velocidad de las partes no críticas de sus programas, y estos intentos de eficiencia en realidad tienen un fuerte impacto negativo cuando se consideran la depuración y el mantenimiento.
>
> — Donald Knuth

Antes de que pueda hacer que su código sea más rápido, primero debe averiguar qué lo hace lento. Esto suena fácil, pero no lo es. Incluso los programadores experimentados tienen dificultades para identificar cuellos de botella en su código. Entonces, en lugar de confiar en su intuición, debe **perfilar** su código: mida el tiempo de ejecución de cada línea de código usando entradas realistas.

Una vez que haya identificado los cuellos de botella, deberá experimentar cuidadosamente con alternativas para encontrar un código más rápido que aún sea equivalente. En el Chapter 24 aprenderá un montón de formas de acelerar el código, pero primero necesita aprender cómo **microbenchmark** para que pueda medir con precisión la diferencia en el rendimiento.

Estructura

- La Section 23.2 le muestra cómo usar las herramientas de creación de perfiles para profundizar exactamente en lo que hace que el código sea lento.
- La Section 23.3 muestra cómo usar microbenchmarking para explorar implementaciones alternativas y descubrir exactamente cuál es la más rápida.

Requisitos

Usaremos profvis para creación de perfiles y bench para microbenchmarking.

```
library(profvis)
library(bench)
```

23.2. Perfiles

En todos los lenguajes de programación, la herramienta principal utilizada para comprender el rendimiento del código es el generador de perfiles. Hay varios tipos diferentes de perfiladores, pero R usa un tipo bastante simple llamado perfilador estadístico o de muestreo. Un generador de perfiles de muestreo detiene la ejecución del código cada pocos milisegundos y registra la pila de llamadas (es decir, qué función se está ejecutando actualmente y la función que llamó a la función, etc.). Por ejemplo, considere `f()`, a continuación:

```
f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}
```

(Utilizo `profvis::pause()` en lugar de `Sys.sleep()` porque `Sys.sleep()` no aparece en las salidas de creación de perfiles porque, por lo que R puede decir, no consume tiempo de cálculo .)

Si perfiláramos la ejecución de `f()`, deteniendo la ejecución del código cada 0.1 s, veríamos un perfil como este:

```
"pause" "f"
"pause" "g" "f"
"pause" "h" "g" "f"
"pause" "h" "f"
```

Cada línea representa un "tick" del generador de perfiles (0,1 s en este caso), y las llamadas a funciones se registran de derecha a izquierda: la primera línea muestra `f()` llamando a `pause()`. Muestra que el código gasta 0.1 s ejecutando `f()`, luego 0.2 s ejecutando `g()`, luego 0.1 s ejecutando `h()`.

Si realmente perfilamos `f()`, usando `utils::Rprof()` como en el código de abajo, es poco probable que obtengamos un resultado tan claro.

```
tmp <- tempfile()
Rprof(tmp, interval = 0.1)
f()
Rprof(NULL)
writeLines(readLines(tmp))
#> sample.interval=100000
#> "pause" "g" "f"
#> "pause" "h" "g" "f"
#> "pause" "h" "f"
```

Esto se debe a que todos los perfiladores deben hacer un equilibrio fundamental entre precisión y rendimiento. El compromiso que se obtiene al usar un generador de perfiles de muestreo solo tiene un impacto mínimo en el rendimiento, pero es fundamentalmente estocástico porque existe cierta variabilidad tanto en la precisión del temporizador como en el tiempo que toma cada operación. Eso significa que cada vez que hagas un perfil obtendrás una respuesta ligeramente diferente. Afortunadamente, la variabilidad afecta más a las funciones que tardan muy poco en ejecutarse, que también son las funciones de menor interés.

23.2.1. Visualización de perfiles

La resolución de creación de perfiles predeterminada es bastante pequeña, por lo que si su función tarda incluso unos segundos, generará cientos de muestras. Eso crece rápidamente más allá de nuestra capacidad de mirar directamente, así que en lugar de usar `utils::Rprof()` usaremos el paquete profvis para visualizar agregados. profvis también conecta los datos de creación de perfiles con el código fuente subyacente, lo que facilita la creación de un modelo mental de lo que necesita cambiar. Si encuentra que profvis no ayuda con su código, puede probar una de las otras opciones como `utils::summaryRprof()` o el paquete proftools (Tierney and Jarjour 2016).

Hay dos formas de usar profvis:

- Desde el menú Profile en RStudio.
- Con `profvis::profvis()`. Recomiendo almacenar su código en un archivo separado y `source()`; esto garantizará que obtenga la mejor conexión entre los datos de creación de perfiles y el código fuente.

```
source("profiling-example.R")
profvis(f())
```

Una vez completada la creación de perfiles, profvis abrirá un documento HTML interactivo que le permitirá explorar los resultados. Hay dos paneles, como se muestra en la Figure 23.1.

El panel superior muestra el código fuente, superpuesto con gráficos de barras para memoria y tiempo de ejecución para cada línea de código. Aquí me centraré en el tiempo y volveremos a la memoria en breve. Esta pantalla le brinda una buena idea general de los cuellos de botella, pero no siempre lo ayuda a identificar con precisión la causa. Aquí, por ejemplo, puedes ver que `h()` tarda 150 ms, el doble que `g()`; eso no se debe a que la función sea más lenta, sino a que se llama con el doble de frecuencia.

El panel inferior muestra un **gráfico de llamas** que muestra la pila de llamadas completa. Esto le permite ver la secuencia completa de llamadas que conducen a cada función, lo que le permite ver que `h()` se llama desde dos lugares diferentes. En esta pantalla, puede pasar el mouse sobre las llamadas individuales para obtener más información y ver la línea correspondiente del código fuente, como en la Figure 23.2.

Alternativamente, puede usar la **pestaña de datos**, Figure 23.3 le permite sumergirse de forma interactiva en el árbol de datos de rendimiento. Esta es básicamente la misma pantalla que el gráfico de llama (girado 90 grados), pero es más útil cuando tiene pilas de llamadas muy grandes o muy anidadas porque puede optar por hacer zoom de forma interactiva solo en los componentes seleccionados.

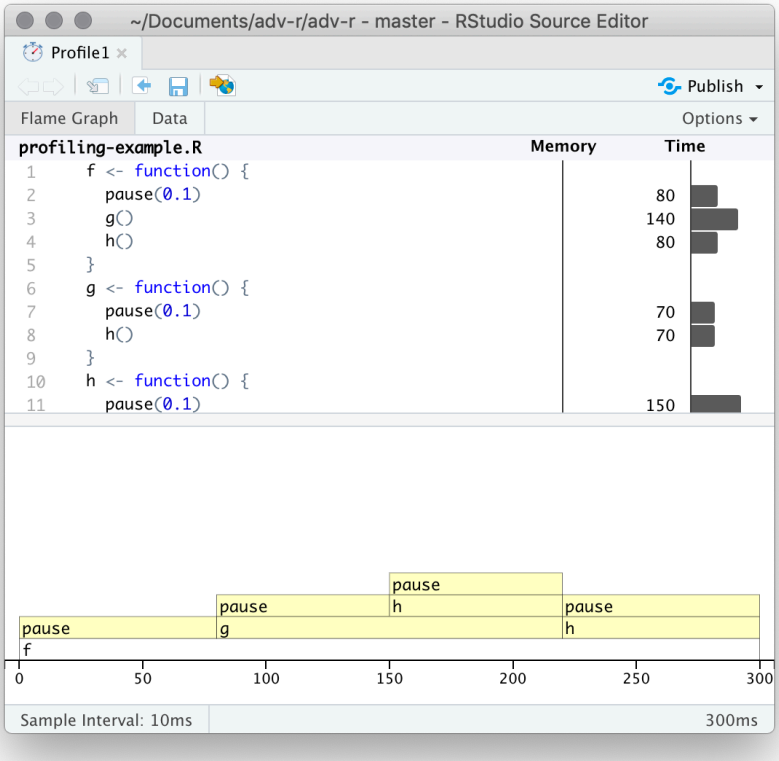


Figure 23.1.: Salida profvis que muestra la fuente en la parte superior y el gráfico de llamas a continuación.

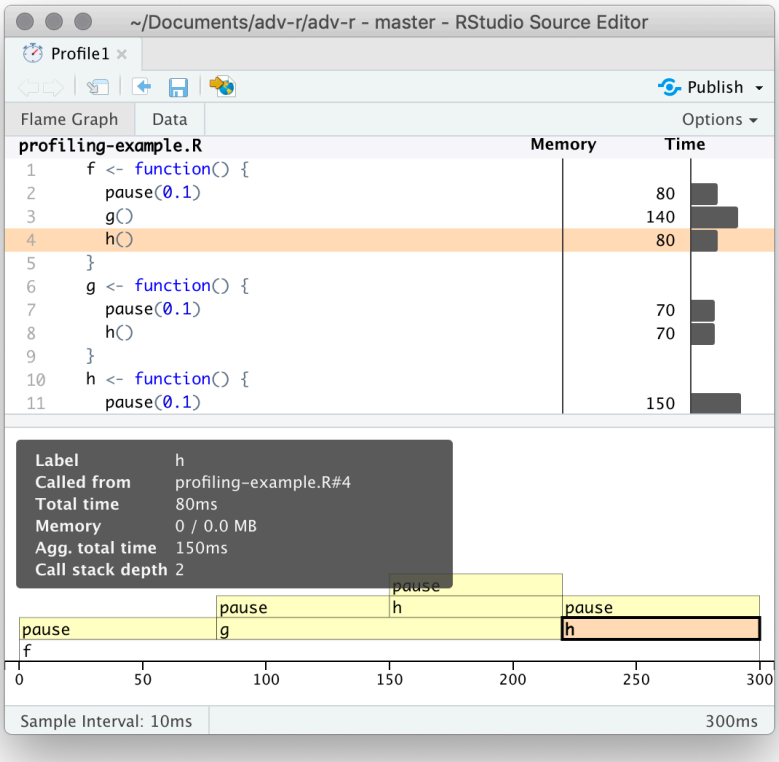


Figure 23.2.: Al pasar el cursor sobre una llamada en el gráfico de llamas, se resalta la línea de código correspondiente y se muestra información adicional sobre el rendimiento.

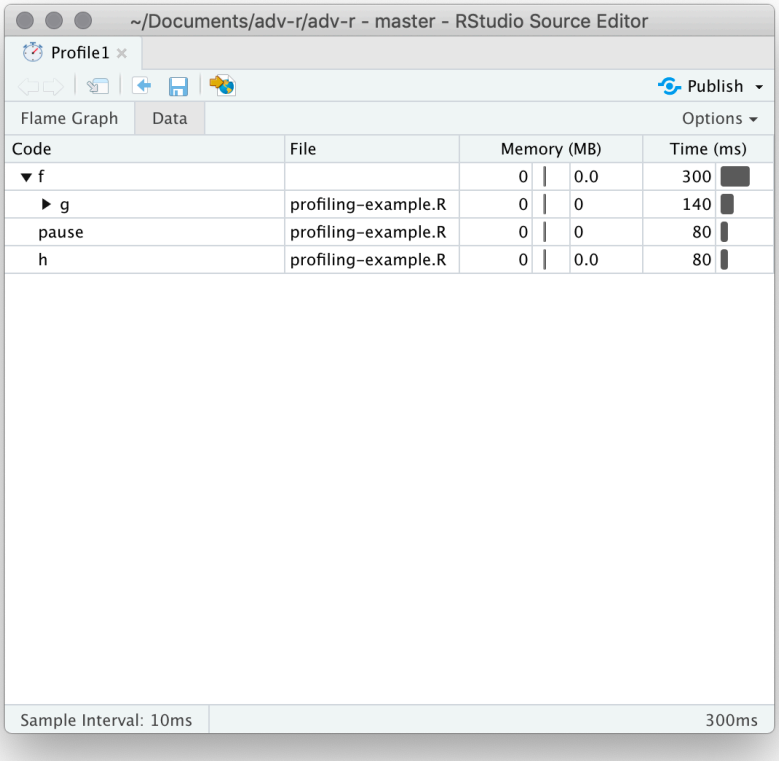


Figure 23.3.: The data gives an interactive tree that allows you to selectively zoom into key components

23.2.2. Perfilado de memoria

Hay una entrada especial en el gráfico de llamas que no corresponde a su código: `<GC>`, que indica que el recolector de basura se está ejecutando. Si `<GC>` toma mucho tiempo, generalmente es una indicación de que está creando muchos objetos de corta duración. Por ejemplo, tome este pequeño fragmento de código:

```
x <- integer()
for (i in 1:1e4) {
  x <- c(x, i)
}
```

Si lo perfilas, verás que la mayor parte del tiempo lo pasa en el recolector de basura, Figure 23.4.

Cuando vea que el recolector de elementos no utilizados ocupa mucho tiempo en su propio código, a menudo puede descubrir el origen del problema observando la columna de memoria: verá una línea donde se asignan grandes cantidades de memoria (el barra de la derecha) y liberada (la barra de la izquierda). Aquí surge el problema debido a la copia al modificar (Section 2.3): cada iteración del bucle crea otra copia de `x`. Aprenderá estrategias para resolver este tipo de problema en la Section 24.6.

23.2.3. Limitaciones

Hay algunas otras limitaciones para la creación de perfiles:

- La generación de perfiles no se extiende al código C. Puede ver si su código R llama al código C/C++ pero no qué funciones se llaman dentro de su código C/C++. Desafortunadamente, las herramientas para generar perfiles de código compilado están más allá del alcance de este libro; Comience mirando https://github.com/r-prof/jointprof.

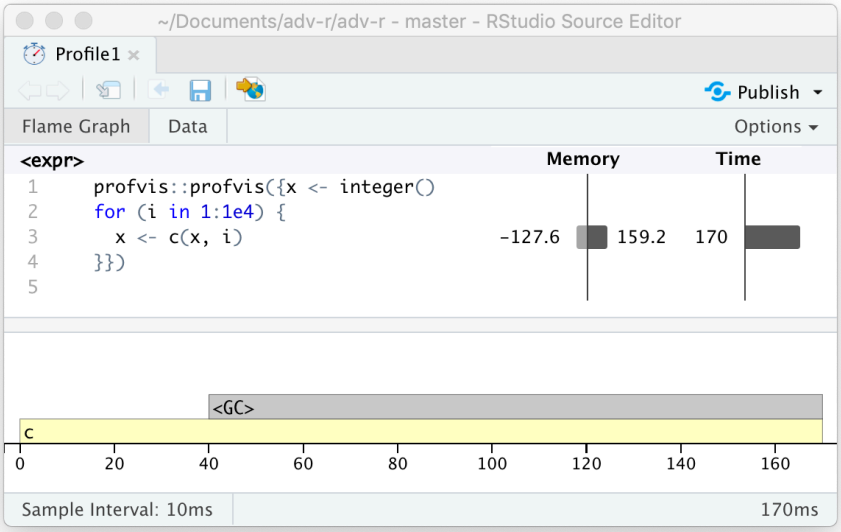


Figure 23.4.: Perfilar un bucle que modifica una variable existente revela que la mayor parte del tiempo se pasa en el recolector de basura().

- Si está haciendo mucha programación funcional con funciones anónimas, puede ser difícil averiguar exactamente qué función se está llamando. La forma más fácil de evitar esto es nombrar sus funciones.

- La evaluación perezosa significa que los argumentos a menudo se evalúan dentro de otra función, y esto complica la pila de llamadas (Section 7.5.2). Desafortunadamente, el generador de perfiles de R no almacena suficiente información para desenredar la evaluación perezosa, por lo que en el siguiente código, la generación de perfiles haría que pareciera que `i()` fue llamado por `j()` porque el argumento no se evalúa hasta que lo necesita `j()`.

```
i <- function() {
  pause(0.1)
  10
}
j <- function(x) {
  x + 10
}
j(i())
```

  Si esto es confuso, use `force()` (Section 10.2.3) para forzar que el cálculo ocurra antes.

23.2.4. Ejercicios

1. Perfila la siguiente función con `torture = 10`. ¿Qué es sorprendente? Lea el código fuente de `rm()` para averiguar qué está pasando.

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

23.3. Microbenchmark

Un **microbenchmark** es una medida del rendimiento de un fragmento de código muy pequeño, algo que puede tardar milisegundos (ms), microsegundos (µs) o nanosegundos (ns) en ejecutarse. Los microbenchmarks son útiles para comparar pequeños fragmentos de código para tareas específicas. Tenga mucho cuidado al generalizar los resultados de los micropuntos de referencia al código real: las diferencias observadas en los micropuntos de referencia normalmente estarán dominadas por efectos de orden superior en el código real; una comprensión profunda de la física subatómica no es muy útil al hornear.

Una gran herramienta para microbenchmarking en R es el paquete de banco (Hester 2018). El paquete de banco utiliza un temporizador de alta precisión, lo que permite comparar operaciones que solo toman una pequeña cantidad de tiempo. Por ejemplo, el siguiente código compara la velocidad de dos enfoques para calcular una raíz cuadrada.

```
x <- runif(100)
(lb <- bench::mark(
  sqrt(x),
  x ^ 0.5
))
#> # A tibble: 2 × 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 sqrt(x)    379.98ns 401.05ns  2317687.      848B     232.
#> 2 x^0.5        1.96µs   2.02µs   481693.      848B       0
```

De forma predeterminada, `bench::mark()` ejecuta cada expresión al menos una vez (`min_iterations = 1`) y, como máximo, las veces necesarias para tardar 0,5 s (`min_time = 0,5`). Comprueba que cada

ejecución devuelve el mismo valor, que suele ser lo que desea microbenchmarking; si desea comparar la velocidad de las expresiones que devuelven valores diferentes, configure `check = FALSE`.

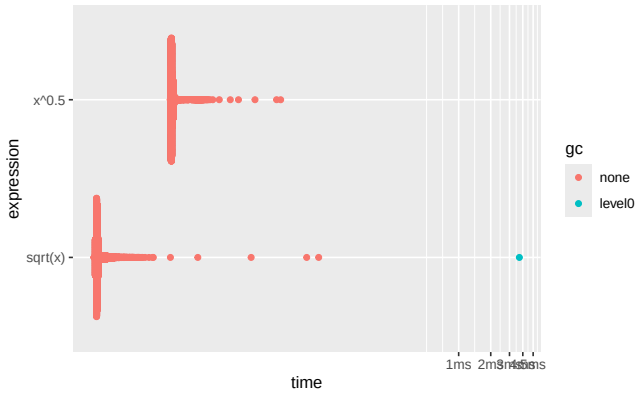
23.3.1. Resultados de `bench::mark()`

`bench::mark()` devuelve los resultados como un tibble, con una fila para cada expresión de entrada y las siguientes columnas:

- `min`, `mean`, `median`, `max`, y `itr/sec` resume el tiempo que tarda la expresión. Concéntrese en el mínimo (el mejor tiempo de ejecución posible) y la mediana (el tiempo típico). En este ejemplo, puede ver que usar la función `sqrt()` de propósito especial es más rápido que el operador de exponenciación general.

  Puedes visualizar la distribución de los tiempos individuales con `plot()`:

```
plot(lb)
#> Loading required namespace: tidyr
```



La distribución tiende a ser muy sesgada hacia la derecha (¡tenga en cuenta que el eje x ya está en una escala logarítmica!), razón por la cual debe evitar comparar medias. También verá a menudo multimodalidad porque su computadora está ejecutando algo más en segundo plano.

- `mem_alloc` te dice la cantidad de memoria asignada por la primera ejecución, y `n_gc()` te dice el número total de recolecciones de basura en todas las ejecuciones. Estos son útiles para evaluar el uso de memoria de la expresión.
- `n_itr` y `total_time` le dice cuántas veces se evaluó la expresión y cuánto tiempo tomó en total. `n_itr` siempre será mayor que el parámetro `min_iteration`, y `total_time` siempre será mayor que el parámetro `min_time`.
- `result`, `memory`, `time`, y `gc` son columnas de lista que almacenan los datos subyacentes sin procesar.

Debido a que el resultado es un tipo especial de tibble, puede usar `[` para seleccionar solo las columnas más importantes. Lo haré con frecuencia en el próximo capítulo.

```
lb[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 × 4
#>   expression      min   median `itr/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl>
#> 1 sqrt(x)    379.98ns 401.05ns  2317687.
#> 2 x^0.5        1.96µs   2.02µs   481693.
```

23.3.2. Interpretación de resultados

Al igual que con todos los micropuntos de referencia, preste mucha atención a las unidades: aquí, cada cálculo toma alrededor de 380 ns, 380 mil

millonésimas de segundo. Para ayudar a calibrar el impacto de un micropunto de referencia en el tiempo de ejecución, es útil pensar cuántas veces debe ejecutarse una función antes de que tarde un segundo. Si un microbenchmark toma:

- 1 ms, entonces mil llamadas toman un segundo.
- 1 µs, luego un millón de llamadas toman un segundo.
- 1 ns, luego mil millones de llamadas toman un segundo.

La función `sqrt()` toma aproximadamente 380 ns, o 0.38 µs, para calcular las raíces cuadradas de 100 números. Eso significa que si repitió la operación un millón de veces, tomaría 0.38 s y, por lo tanto, es poco probable que cambiar la forma en que calcula la raíz cuadrada afecte significativamente el código real. Esta es la razón por la que debe tener cuidado al generalizar los resultados de microbenchmarking.

23.3.3. Ejercicios

1. En lugar de usar `bench::mark()`, podrías usar la función integrada `system.time()`. Pero `system.time()` es mucho menos preciso, por lo que deberá repetir cada operación muchas veces con un ciclo y luego dividir para encontrar el tiempo promedio de cada operación, como en el código a continuación.

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

   ¿Cómo se comparan las estimaciones de `system.time()` con las de `bench::mark()`? ¿Por qué son diferentes?

2. Aquí hay otras dos formas de calcular la raíz cuadrada de un vector. ¿Cuál crees que será más rápido? ¿Cuál será más lento? Utilice microbenchmarking para probar sus respuestas.

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

24. Mejorando el desempeño

24.1. Introducción

> Deberíamos olvidarnos de las pequeñas eficiencias, digamos alrededor del 97% del tiempo: la optimización prematura es la raíz de todos los males. Sin embargo, no debemos dejar pasar nuestras oportunidades en ese crítico 3%. Un buen programador no se dejará llevar por la complacencia de tal razonamiento, será prudente al mirar cuidadosamente el código crítico; pero solo después de que ese código haya sido identificado.
>
> — Donald Knuth

Una vez que haya utilizado la creación de perfiles para identificar un cuello de botella, debe hacerlo más rápido. Es difícil dar consejos generales sobre cómo mejorar el rendimiento, pero hago lo mejor que puedo con cuatro técnicas que se pueden aplicar en muchas situaciones. También sugeriré una estrategia general para la optimización del rendimiento que ayude a garantizar que su código más rápido siga siendo correcto.

Es fácil quedar atrapado tratando de eliminar todos los cuellos de botella. ¡No! Su tiempo es valioso y es mejor gastarlo analizando sus datos, no eliminando posibles ineficiencias en su código. Sea pragmático: no gaste horas de su tiempo para ahorrar segundos de tiempo de computadora. Para hacer cumplir este consejo, debe establecer un objetivo de tiempo para su código y optimizar solo hasta ese objetivo. Esto significa que no

eliminará todos los cuellos de botella. Algunas no las alcanzarás porque has cumplido tu objetivo. Es posible que deba pasar por alto otros y aceptarlos porque no hay una solución rápida y fácil o porque el código ya está bien optimizado y no es posible una mejora significativa. Acepte estas posibilidades y pase al siguiente candidato.

Si desea obtener más información sobre las características de rendimiento del lenguaje R, le recomiendo *Evaluar el diseño del lenguaje R* (Morandat et al. 2012). Saca conclusiones al combinar un intérprete R modificado con un amplio conjunto de código que se encuentra en la naturaleza.

Estructura

- La Section 24.2 le enseña cómo organizar su código para que la optimización sea lo más fácil y libre de errores posible.
- La Section 24.3 le recuerda que busque las soluciones existentes.
- La Section 24.4 enfatiza la importancia de ser perezoso: a menudo, la forma más fácil de hacer una función más rápida es dejar que haga menos trabajo.
- La Section 24.5 define de forma concisa la vectorización y le muestra cómo aprovechar al máximo las funciones integradas.
- La Section 24.6 analiza los peligros de rendimiento de la copia de datos.
- La Section 24.7 reúne todas las piezas en un estudio de caso que muestra cómo acelerar las pruebas t repetidas unas mil veces.
- La Section 24.8 termina el capítulo con indicaciones a más recursos que lo ayudarán a escribir código rápido.

Requisitos previos

Usaremos bench para comparar con precisión el rendimiento de pequeños fragmentos de código independientes.

```
library(bench)
```

24.2. Organización del código

Hay dos trampas en las que es fácil caer cuando intentas hacer tu código más rápido:

1. Escribir código más rápido pero incorrecto.
2. Escribir código que crees que es más rápido, pero que en realidad no es mejor.

La estrategia descrita a continuación le ayudará a evitar estas trampas.

Al abordar un cuello de botella, es probable que encuentre múltiples enfoques. Escriba una función para cada enfoque, encapsulando todo el comportamiento relevante. Esto hace que sea más fácil verificar que cada enfoque devuelva el resultado correcto y cronometrar cuánto tiempo lleva ejecutarse. Para demostrar la estrategia, compararé dos enfoques para calcular la media:

```
mean1 <- function(x) mean(x)
mean2 <- function(x) sum(x) / length(x)
```

Te recomiendo que lleves un registro de todo lo que intentes, incluso de los fracasos. Si ocurre un problema similar en el futuro, será útil ver todo lo que ha intentado. Para hacer esto, recomiendo RMarkdown, que facilita la combinación de código con comentarios y notas detallados.

A continuación, genere un caso de prueba representativo. El caso debe ser lo suficientemente grande para capturar la esencia de su problema, pero lo suficientemente pequeño como para que solo tome unos segundos como máximo. No desea que tarde demasiado porque necesitará ejecutar el caso de prueba muchas veces para comparar enfoques. Por otro lado, no desea que el caso sea demasiado pequeño porque es posible que los resultados no alcancen el problema real. Aquí voy a usar 100,000 números:

```
x <- runif(1e5)
```

Ahora usa `bench::mark()` para comparar con precisión las variaciones. `bench::mark()` verifica automáticamente que todas las llamadas devuelvan los mismos valores. Esto no garantiza que la función se comporte de la misma manera para todas las entradas, por lo que en un mundo ideal también tendrá pruebas unitarias para asegurarse de no cambiar accidentalmente el comportamiento de la función.

```
bench::mark(
  mean1(x),
  mean2(x)
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 × 4
#>   expression      min   median `itr/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl>
#> 1 mean1(x)      379µs    437µs     2277.
#> 2 mean2(x)      187µs    187µs     5311.
```

(Puede que te sorprendan los resultados: `mean(x)` es considerablemente más lento que `sum(x) / length(x)`. Esto se debe a que, entre otras razones, `mean(x)` hace dos pasadas sobre el vector para que sea numéricamente más preciso.)

Si desea ver esta estrategia en acción, la he usado varias veces en stackoverflow:

- http://stackoverflow.com/questions/22515525#22518603
- http://stackoverflow.com/questions/22515175#22515856
- http://stackoverflow.com/questions/3476015#22511936

24.3. Comprobación de soluciones existentes

Una vez que haya organizado su código y capturado todas las variaciones que se le ocurran, es natural ver lo que otros han hecho. Eres parte de una gran comunidad y es muy posible que alguien ya haya abordado el mismo problema. Dos buenos lugares para comenzar son:

- CRAN task views. Si hay una vista de tareas CRAN relacionada con el dominio de su problema, vale la pena mirar los paquetes enumerados allí.

- Dependencias inversas de Rcpp, como se indica en su página de CRAN. Dado que estos paquetes usan C++, es probable que sean rápidos.

De lo contrario, el desafío es describir su cuello de botella de una manera que lo ayude a encontrar problemas y soluciones relacionados. Saber el nombre del problema o sus sinónimos hará que esta búsqueda sea mucho más fácil. Pero como no sabes cómo se llama, ¡es difícil buscarlo! La mejor manera de resolver este problema es leer mucho para que puedas construir tu propio vocabulario con el tiempo. Alternativamente, pregunte a otros. Hable con sus colegas y haga una lluvia de ideas sobre algunos nombres posibles, luego busque en Google y StackOverflow. Suele ser útil restringir la búsqueda a páginas relacionadas con R. Para Google, pruebe rseek. Para stackoverflow, restrinja su búsqueda incluyendo la etiqueta R, `[R]`, en su búsqueda.

Registre todas las soluciones que encuentre, no solo aquellas que parezcan ser más rápidas inmediatamente. Algunas soluciones pueden ser más

lentas inicialmente, pero terminan siendo más rápidas porque son más fáciles de optimizar. También puede combinar las partes más rápidas desde diferentes enfoques. Si ha encontrado una solución lo suficientemente rápida, ¡felicidades! De lo contrario, sigue leyendo.

24.3.1. Ejercicios

1. ¿Cuáles son las alternativas más rápidas a `lm()`? ¿Cuáles están diseñados específicamente para trabajar con conjuntos de datos más grandes?

2. ¿Qué paquete implementa una versión de `match()` que es más rápida para búsquedas repetidas? ¿Cuánto más rápido es?

3. Enumere cuatro funciones (no solo las de base R) que convierten una cadena en un objeto de fecha y hora. Cuales son sus fortalezas y debilidades?

4. ¿Qué paquetes brindan la capacidad de calcular una media móvil?

5. ¿Cuáles son las alternativas a `optim()`?

24.4. Haciendo lo menos posible

La forma más fácil de hacer que una función sea más rápida es dejar que haga menos trabajo. Una forma de hacerlo es usar una función adaptada a un tipo de entrada o salida más específico, o a un problema más específico. Por ejemplo:

- `rowSums()`, `colSums()`, `rowMeans()`, y `colMeans()` son más rápidas que las invocaciones equivalentes que usan `apply()` porque están vectorizadas (Section 24.5).

- `vapply()` es más rápido que `sapply()` porque especifica previamente el tipo de salida.
- Si quiere ver si un vector contiene un solo valor, `any(x == 10)` es mucho más rápido que `10 %in% x` porque probar la igualdad es más simple que probar la inclusión de conjuntos.

Tener este conocimiento al alcance de la mano requiere saber que existen funciones alternativas: es necesario tener un buen vocabulario. Amplíe su vocabulario leyendo regularmente el código R. Buenos lugares para leer código son la lista de correo de R-help y StackOverflow.

Algunas funciones obligan a sus entradas a un tipo específico. Si su entrada no es del tipo correcto, la función tiene que hacer un trabajo extra. En su lugar, busque una función que funcione con sus datos tal como están, o considere cambiar la forma en que almacena sus datos. El ejemplo más común de este problema es usar `apply()` en un marco de datos. `apply()` siempre convierte su entrada en una matriz. No solo es propenso a errores (porque un marco de datos es más general que una matriz), sino que también es más lento.

Otras funciones harán menos trabajo si les proporciona más información sobre el problema. Siempre vale la pena leer detenidamente la documentación y experimentar con diferentes argumentos. Algunos ejemplos que he descubierto en el pasado incluyen:

- `read.csv()`: especificar tipos de columnas conocidas con `colClasses`. (También considere cambiar a `readr::read_csv()` o `data.table::fread()` que son considerablemente más rápidos que `read.csv()`.)
- `factor()`: especificar niveles conocidos con `levels`.
- `cut()`: no genere etiquetas con `labels = FALSE` si no las necesita o, mejor aún, use `findInterval()` como se menciona en la sección "ver también" de la documentación.

- `unlist(x, use.names = FALSE)` es mucho más rápido que `unlist(x)`.
- `interaction()`: si solo necesita combinaciones que existen en los datos, use `drop = TRUE`.

A continuación, exploro cómo podría mejorar la aplicación de esta estrategia para mejorar el rendimiento de `mean()` y `as.data.frame()`.

24.4.1. `mean()`

A veces, puede hacer que una función sea más rápida evitando el envío de métodos. Si está llamando a un método en un ciclo cerrado, puede evitar algunos de los costos haciendo la búsqueda del método solo una vez:

- Para S3, puede hacer esto llamando a `generic.class()` en lugar de `generic()`.
- Para S4, puede hacer esto usando `selectMethod()` para encontrar el método, guardándolo en una variable y luego llamando a esa función.

Por ejemplo, llamar a `mean.default()` es un poco más rápido que llamar a `mean()` para vectores pequeños:

```
x <- runif(1e2)

bench::mark(
  mean(x),
  mean.default(x)
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 × 4
#>   expression           min   median `itr/sec`
#>   <bch:expr>      <bch:tm> <bch:tm>     <dbl>
#> 1 mean(x)              3µs   3.29µs   294876.
#> 2 mean.default(x)   1.89µs      2µs   477924.
```

Esta optimización es un poco arriesgada. Si bien `mean.default()` es casi el doble de rápido para 100 valores, fallará de manera sorprendente si `x` no es un vector numérico.

Una optimización aún más arriesgada es llamar directamente a la función `.Internal` subyacente. Esto es más rápido porque no realiza ninguna verificación de entrada ni maneja NA, por lo que está comprando velocidad a costa de la seguridad.

```
x <- runif(1e2)
bench::mark(
  mean(x),
  mean.default(x),
  .Internal(mean(x))
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 3 × 4
#>   expression              min   median `itr/sec`
#>   <bch:expr>         <bch:tm> <bch:tm>     <dbl>
#> 1 mean(x)                3µs   3.29µs   296903.
#> 2 mean.default(x)      1.88µs   1.99µs   484725.
#> 3 .Internal(mean(x)) 480.91ns 500.94ns  1950263.
```

NB: La mayoría de estas diferencias surgen porque `x` es pequeño. Si aumenta el tamaño, las diferencias básicamente desaparecen, porque la mayor parte del tiempo ahora se dedica a calcular la media, sin encontrar la implementación subyacente. Este es un buen recordatorio de que el tamaño de la entrada es importante y debe motivar sus optimizaciones en función de datos realistas.

```
x <- runif(1e4)
bench::mark(
  mean(x),
  mean.default(x),
  .Internal(mean(x))
```

```
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 3 × 4
#>   expression             min   median `itr/sec`
#>   <bch:expr>         <bch:tm> <bch:tm>     <dbl>
#> 1 mean(x)              41.6µs   46.1µs    21571.
#> 2 mean.default(x)        41µs   44.8µs    22194.
#> 3 .Internal(mean(x))   37.6µs   43.3µs    23034.
```

24.4.2. `as.data.frame()`

Saber que está tratando con un tipo específico de entrada puede ser otra forma de escribir código más rápido. Por ejemplo, `as.data.frame()` es bastante lento porque convierte cada elemento en un marco de datos y luego `rbind()` los une. Si tiene una lista con nombre con vectores de igual longitud, puede transformarla directamente en un marco de datos. En este caso, si puede hacer suposiciones sólidas sobre su entrada, puede escribir un método que sea considerablemente más rápido que el predeterminado.

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

l <- lapply(1:26, function(i) runif(1e3))
names(l) <- letters

bench::mark(
  as.data.frame = as.data.frame(l),
  quick_df      = quickdf(l)
)[c("expression", "min", "median", "itr/sec", "n_gc")]
```

```
#> # A tibble: 2 × 4
#>   expression         min   median `itr/sec`
#>   <bch:expr>    <bch:tm> <bch:tm>     <dbl>
#> 1 as.data.frame 948.52µs   1.01ms      975.
#> 2 quick_df        6.25µs   7.27µs   126808.
```

Una vez más, tenga en cuenta la compensación. Este método es rápido porque es peligroso. Si le da entradas incorrectas, obtendrá un marco de datos corrupto:

```
quickdf(list(x = 1, y = 1:2))
#> Warning in format.data.frame(if (omit) x[seq_len(n0), , drop =
#> FALSE] else x, : corrupt data frame: columns will be truncated or
#> padded with NAs
#>   x y
#> 1 1 1
```

Para llegar a este método mínimo, leí cuidadosamente y luego reescribí el código fuente para `as.data.frame.list()` y `data.frame()`. Hice muchos pequeños cambios, comprobando cada vez que no había roto el comportamiento existente. Después de varias horas de trabajo, pude aislar el código mínimo que se muestra arriba. Esta es una técnica muy útil. La mayoría de las funciones básicas de R están escritas para la flexibilidad y la funcionalidad, no para el rendimiento. Por lo tanto, reescribir para su necesidad específica a menudo puede generar mejoras sustanciales. Para hacer esto, deberá leer el código fuente. Puede ser complejo y confuso, ¡pero no te rindas!

24.4.3. Ejercicios

1. ¿Cuál es la diferencia entre `rowSums()` y `.rowSums()`?

2. Cree una versión más rápida de `chisq.test()` que solo calcula la estadística de prueba de chi-cuadrado cuando la entrada son dos vectores numéricos sin valores faltantes. Puede intentar simplificar `chisq.test()` o codificar desde la definición matemática.

3. ¿Puedes hacer una versión más rápida de `table()` para el caso de una entrada de dos vectores enteros sin valores perdidos? ¿Puedes usarlo para acelerar tu prueba de chi-cuadrado?

24.5. Vectorizar

Si ha usado R durante algún tiempo, probablemente haya escuchado la advertencia de "vectorizar su código". Pero, ¿qué significa eso realmente? Vectorizar su código no se trata solo de evitar bucles for, aunque eso suele ser un paso. Vectorizar se trata de adoptar un enfoque de objeto completo para un problema, pensando en vectores, no en escalares. Hay dos atributos clave de una función vectorizada:

- Simplifica muchos problemas. En lugar de tener que pensar en los componentes de un vector, solo piensa en vectores completos.

- Los bucles en una función vectorizada están escritos en C en lugar de R. Los bucles en C son mucho más rápidos porque tienen mucha menos sobrecarga.

El Chapter 9 hizo hincapié en la importancia del código vectorizado como una abstracción de mayor nivel. La vectorización también es importante para escribir código R rápido. Esto no significa simplemente usar `map()` o `lapply()`. En cambio, la vectorización significa encontrar la función R existente que se implementa en C y se aplica más a su problema.

Las funciones vectorizadas que se aplican a muchos cuellos de botella de rendimiento comunes incluyen:

- `rowSums()`, `colSums()`, `rowMeans()`, y `colMeans()`. Estas funciones matriciales vectorizadas siempre serán más rápidas que usar `apply()`. A veces puede usar estas funciones para construir otras funciones vectorizadas.

```
rowAny <- function(x) rowSums(x) > 0
rowAll <- function(x) rowSums(x) == ncol(x)
```

- La creación de subconjuntos vectorizados puede conducir a grandes mejoras en la velocidad. Recuerde las técnicas detrás de las tablas de búsqueda (Section 4.5.1) y la combinación y combinación manual (Section 4.5.2). Recuerde también que puede usar la asignación de subconjuntos para reemplazar varios valores en un solo paso. Si `x` es un vector, una matriz o un marco de datos, entonces `x[is.na(x)] <- 0` reemplazará todos los valores faltantes con 0.
- Si está extrayendo o reemplazando valores en ubicaciones dispersas en una matriz o marco de datos, subconjunto con una matriz de enteros. Consulte Section 4.2.3 para obtener más detalles.
- Si está convirtiendo valores continuos a categóricos, asegúrese de saber cómo usar `cut()` y `findInterval()`.
- Tenga en cuenta las funciones vectorizadas como `cumsum()` y `diff()`.

El álgebra matricial es un ejemplo general de vectorización. Estos bucles son ejecutados por bibliotecas externas altamente optimizadas como BLAS. Si puede encontrar una manera de usar el álgebra matricial para resolver su problema, a menudo obtendrá una solución muy rápida. La habilidad para resolver problemas con álgebra matricial es producto de la experiencia. Un buen lugar para comenzar es preguntar a personas con experiencia en su dominio.

La vectorización tiene un inconveniente: es más difícil predecir cómo escalarán las operaciones. El siguiente ejemplo mide cuánto tiempo lleva usar subconjuntos de caracteres para buscar 1, 10 y 100 elementos de una lista. Podría esperar que buscar 10 elementos tomara 10 veces más que buscar

1, y que buscar 100 elementos tomaría 10 veces más de nuevo. De hecho, el siguiente ejemplo muestra que solo se tarda aproximadamente ~10 veces más en buscar 100 elementos que en buscar 1. Eso sucede porque una vez que llega a un cierto tamaño, la implementación interna cambia a una estrategia que tiene un mayor costo de instalación, pero escala mejor.

```
lookup <- setNames(as.list(sample(100, 26)), letters)

x1 <- "j"
x10 <- sample(letters, 10)
x100 <- sample(letters, 100, replace = TRUE)

bench::mark(
  lookup[x1],
  lookup[x10],
  lookup[x100],
  check = FALSE
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 3 × 4
#>   expression        min   median `itr/sec`
#>   <bch:expr>   <bch:tm> <bch:tm>     <dbl>
#> 1 lookup[x1]   430.97ns 501.17ns  1796467.
#> 2 lookup[x10]    1.25µs   1.35µs   708770.
#> 3 lookup[x100]   2.88µs   4.69µs   215670.
```

La vectorización no resolverá todos los problemas y, en lugar de convertir un algoritmo existente en uno que utilice un enfoque vectorizado, a menudo es mejor escribir su propia función vectorizada en C++. Aprenderá cómo hacerlo en el Chapter 25.

24.5.1. Ejercicios

1. Las funciones de densidad, por ejemplo, `dnorm()`, tienen una interfaz común. ¿Qué argumentos se vectorizan? ¿Qué hace `rnorm(10, mean = 10:1)`?

2. Compara la velocidad de `apply(x, 1, sum)` con `rowSums(x)` para diferentes tamaños de `x`.

3. ¿Cómo puedes usar `crossprod()` para calcular una suma ponderada? ¿Cuánto más rápido es que el ingenuo `sum(x * w)`?

24.6. Evitar copias

Una fuente perniciosa de código R lento es hacer crecer un objeto con un bucle. Siempre que use `c()`, `append()`, `cbind()`, `rbind()` o `paste()` para crear un objeto más grande, R primero debe asignar espacio para el nuevo objeto y luego copiar el objeto antiguo a su nuevo hogar. Si repite esto muchas veces, como en un ciclo for, esto puede ser bastante costoso. Has entrado en el Círculo 2 del *R inferno*.

Viste un ejemplo de este tipo de problema en la Section 23.2.2, así que aquí mostraré un ejemplo un poco más complejo del mismo problema básico. Primero generamos algunas cadenas aleatorias y luego las combinamos iterativamente con un ciclo usando `collapse()`, o en un solo paso usando `paste()`. Tenga en cuenta que el rendimiento de `collapse()` empeora relativamente a medida que aumenta el número de cadenas: combinar 100 cadenas lleva casi 30 veces más que combinar 10 cadenas.

```
random_string <- function() {
  paste(sample(letters, 50, replace = TRUE), collapse = "")
}
strings10 <- replicate(10, random_string())
strings100 <- replicate(100, random_string())
```

```
collapse <- function(xs) {
  out <- ""
  for (x in xs) {
    out <- paste0(out, x)
  }
  out
}

bench::mark(
  loop10  = collapse(strings10),
  loop100 = collapse(strings100),
  vec10   = paste(strings10, collapse = ""),
  vec100  = paste(strings100, collapse = ""),
  check = FALSE
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 4 × 4
#>   expression      min   median `itr/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl>
#> 1 loop10      19.67µs  21.23µs    46297.
#> 2 loop100    521.78µs 534.79µs     1847.
#> 3 vec10        3.58µs   3.93µs   247621.
#> 4 vec100      22.54µs  23.95µs    42016.
```

Modificar un objeto en un bucle, por ejemplo, `x[i] <- y`, también puede crear una copia, dependiendo de la clase de `x`. La Section 2.5.1 analiza este problema con mayor profundidad y le brinda algunas herramientas para determinar cuándo está haciendo copias.

24.7. Caso de estudio: t-test

El siguiente estudio de caso muestra cómo hacer que las pruebas t sean más rápidas utilizando algunas de las técnicas descritas anteriormente. Se basa en un ejemplo de *Cálculo de miles de estadísticas de prueba simultáneamente en R* de Holger Schwender y Tina Müller. Recomiendo encarecidamente leer el documento completo para ver la misma idea aplicada a otras pruebas.

Imagine que hemos realizado 1000 experimentos (filas), cada uno de los cuales recopila datos de 50 individuos (columnas). Los primeros 25 individuos de cada experimento se asignan al grupo 1 y el resto al grupo 2. Primero generaremos algunos datos aleatorios para representar este problema:

```
m <- 1000
n <- 50
X <- matrix(rnorm(m * n, mean = 10, sd = 3), nrow = m)
grp <- rep(1:2, each = n / 2)
```

Para los datos en este formulario, hay dos formas de usar `t.test()`. Podemos usar la interfaz de fórmula o proporcionar dos vectores, uno para cada grupo. El tiempo revela que la interfaz de la fórmula es considerablemente más lenta.

```
system.time(
  for (i in 1:m) {
    t.test(X[i, ] ~ grp)$statistic
  }
)
#>    user  system elapsed
#>   0.406   0.000   0.406
system.time(
```

```
  for (i in 1:m) {
    t.test(X[i, grp == 1], X[i, grp == 2])$statistic
  }
)
#>    user  system elapsed
#>   0.114   0.000   0.114
```

Por supuesto, un bucle for calcula, pero no guarda los valores. Podemos `map_dbl()` (Section 9.2.1) para hacer eso. Esto agrega un poco de sobrecarga:

```
compT <- function(i){
  t.test(X[i, grp == 1], X[i, grp == 2])$statistic
}
system.time(t1 <- purrr::map_dbl(1:m, compT))
#>    user  system elapsed
#>   0.126   0.000   0.126
```

¿Cómo podemos hacer esto más rápido? Primero, podríamos intentar hacer menos trabajo. Si observa el código fuente de `stats:::t.test.default()`, verá que hace mucho más que calcular la estadística t. También calcula el valor p y formatea la salida para su impresión. Podemos intentar que nuestro código sea más rápido eliminando esas piezas.

```
my_t <- function(x, grp) {
  t_stat <- function(x) {
    m <- mean(x)
    n <- length(x)
    var <- sum((x - m) ^ 2) / (n - 1)

    list(m = m, n = n, var = var)
  }
```

```
  g1 <- t_stat(x[grp == 1])
  g2 <- t_stat(x[grp == 2])

  se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
  (g1$m - g2$m) / se_total
}

system.time(t2 <- purrr::map_dbl(1:m, ~ my_t(X[.,], grp)))
#>    user  system elapsed
#>   0.024   0.000   0.024
stopifnot(all.equal(t1, t2))
```

Esto nos da una mejora de velocidad de seis veces.

Ahora que tenemos una función bastante simple, podemos hacerla aún más rápida al vectorizarla. En lugar de recorrer la matriz fuera de la función, modificaremos `t_stat()` para que funcione con una matriz de valores. Por lo tanto, `mean()` se convierte en `rowMeans()`, `length()` se convierte en `ncol()` y `sum()` se convierte en `rowSums()`. El resto del código permanece igual.

```
rowtstat <- function(X, grp){
  t_stat <- function(X) {
    m <- rowMeans(X)
    n <- ncol(X)
    var <- rowSums((X - m) ^ 2) / (n - 1)

    list(m = m, n = n, var = var)
  }

  g1 <- t_stat(X[, grp == 1])
  g2 <- t_stat(X[, grp == 2])
```

```
  se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
  (g1$m - g2$m) / se_total
}
system.time(t3 <- rowtstat(X, grp))
#>    user  system elapsed
#>    0.01    0.00    0.01
stopifnot(all.equal(t1, t3))
```

¡Eso es mucho más rápido! Es al menos 40 veces más rápido que nuestro esfuerzo anterior y alrededor de 1000 veces más rápido que donde comenzamos.

24.8. Otras tecnicas

Ser capaz de escribir código R rápido es parte de ser un buen programador R. Más allá de las sugerencias específicas de este capítulo, si desea escribir código R rápido, deberá mejorar sus habilidades generales de programación. Algunas formas de hacer esto son:

- Read R blogs para ver con qué problemas de rendimiento han luchado otras personas y cómo han hecho que su código sea más rápido.
- Lea otros libros de programación R, como *El arte de la programación R* (Matloff 2011) o [*R Inferno*] de Patrick Burns (http://www.burns-stat.com/documents/books/the -r-inferno/) para conocer las trampas comunes.
- Tome un curso de algoritmos y estructura de datos para aprender algunas formas bien conocidas de abordar ciertas clases de problemas. Escuché cosas buenas sobre el curso de algoritmos de Princeton que se ofrece en Coursera.

- Aprende a paralelizar tu código. Dos lugares para comenzar son *Parallel R* (McCallum and Weston 2011) y *Parallel Computing for Data Science* (Matloff 2015).
- Lea libros generales sobre optimización como *Optimización madura* (Bueno 2013) o el *Programador pragmático* (Hunt and Thomas 1990).

También puede comunicarse con la comunidad para obtener ayuda. StackOverflow puede ser un recurso útil. Deberá esforzarse un poco para crear un ejemplo fácilmente digerible que también capture las características más destacadas de su problema. Si su ejemplo es demasiado complejo, pocas personas tendrán el tiempo y la motivación para intentar una solución. Si es demasiado simple, obtendrá respuestas que resuelven el problema del juguete pero no el problema real. Si también intenta responder preguntas en StackOverflow, rápidamente tendrá una idea de lo que constituye una buena pregunta.

25. Reescribiendo código de R en C++

25.1. Introducción

A veces, el código R simplemente no es lo suficientemente rápido. Ha utilizado la creación de perfiles para descubrir dónde están los cuellos de botella y ha hecho todo lo posible en R, pero su código aún no es lo suficientemente rápido. En este capítulo, aprenderá a mejorar el rendimiento reescribiendo funciones clave en C++. Esta magia viene a través del paquete Rcpp (Eddelbuettel and François 2011) (con contribuciones clave de Doug Bates, John Chambers y JJ Allaire).

Rcpp hace que sea muy simple conectar C++ a R. Si bien es *posible* escribir código C o Fortran para usar en R, será doloroso en comparación. Rcpp proporciona una API limpia y accesible que le permite escribir código de alto rendimiento, aislado de la compleja API C de R.

Los cuellos de botella típicos que C++ puede abordar incluyen:

- Bucles que no se pueden vectorizar fácilmente porque las iteraciones posteriores dependen de las anteriores.
- Funciones recursivas, o problemas que implican llamar funciones millones de veces. La sobrecarga de llamar a una función en C++ es mucho menor que en R.

- Problemas que requieren estructuras de datos y algoritmos avanzados que R no proporciona. A través de la biblioteca de plantillas estándar (STL), C++ tiene implementaciones eficientes de muchas estructuras de datos importantes, desde mapas ordenados hasta colas de dos extremos.

El objetivo de este capítulo es discutir solo aquellos aspectos de C++ y Rcpp que son absolutamente necesarios para ayudarlo a eliminar los cuellos de botella en su código. No dedicaremos mucho tiempo a funciones avanzadas como la programación orientada a objetos o las plantillas porque el enfoque está en escribir funciones pequeñas e independientes, no grandes programas. Un conocimiento práctico de C++ es útil, pero no esencial. Muchos buenos tutoriales y referencias están disponibles gratuitamente, incluidos http://www.learncpp.com/ y https://en.cppreference.com/w/cpp. Para temas más avanzados, la serie *Effective C++* de Scott Meyers es una opción popular.

Estructura

- La Section 25.2 teaches you how to write C++ by converting simple R functions to their C++ equivalents. You'll learn how C++ differs from R, and what the key scalar, vector, and matrix classes are called.

- La Section 25.2.5 le muestra cómo usar `sourceCpp()` para cargar un archivo C++ desde el disco de la misma manera que usa `source()` para cargar un archivo de código R.

- La Section 25.3 analiza cómo modificar los atributos de Rcpp y menciona algunas de las otras clases importantes.

- La Section 25.4 le enseña cómo trabajar con los valores faltantes de R en C++.

- La Section 25.5 le muestra cómo usar algunas de las estructuras de datos y algoritmos más importantes de la biblioteca de plantillas estándar, o STL, integrada en C++.
- La Section 25.6 muestra dos estudios de casos reales en los que se utilizó Rcpp para obtener mejoras de rendimiento considerables.
- La Section 25.7 le enseña cómo agregar código C++ a un paquete.
- La Section 25.8 concluye el capítulo con una lista de más recursos para ayudarlo a aprender Rcpp y C++.

Requisitos previos

Usaremos Rcpp para llamar a C++ desde R:

```
library(Rcpp)
```

También necesitará un compilador de C++ que funcione. Para conseguirlo:

- En Windows, instale Rtools.
- En Mac, instala Xcode desde la tienda de aplicaciones.
- En Linux, `sudo apt-get install r-base-dev` o similar.

25.2. Empezar con C++

`cppFunction()` le permite escribir funciones de C++ en R:

```
cppFunction('int add(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}')
# add funciona como una función R normal
add
#> function (x, y, z)
#> .Call(<pointer: 0x7f8c7d0c24d0>, x, y, z)
add(1, 2, 3)
#> [1] 6
```

Cuando ejecute este código, Rcpp compilará el código C++ y construirá una función R que se conecta a la función C++ compilada. Suceden muchas cosas debajo del capó, pero Rcpp se encarga de todos los detalles para que no tengas que preocuparte por ellos.

Las siguientes secciones le enseñarán los conceptos básicos mediante la traducción de funciones simples de R a sus equivalentes de C++. Comenzaremos de manera simple con una función que no tiene entradas y una salida escalar, y luego la complicaremos progresivamente:

- Entrada escalar y salida escalar
- Entrada vectorial y salida escalar
- Entrada vectorial y salida vectorial
- Entrada matricial y salida vectorial

25.2.1. Sin entradas, salida escalar

Comencemos con una función muy simple. No tiene argumentos y siempre devuelve el entero 1:

```
one <- function() 1L
```

The equivalent C++ function is:

```
int one() {
  return 1;
}
```

La podemos compilar y usar desde R con `cppFunction()`

```
cppFunction('int one() {
  return 1;
}')
```

Esta pequeña función ilustra una serie de diferencias importantes entre R y C++:

- La sintaxis para crear una función se parece a la sintaxis para llamar a una función; no usa la asignación para crear funciones como lo hace en R.
- Debe declarar el tipo de salida que devuelve la función. Esta función devuelve un `int` (un entero escalar). Las clases para los tipos más comunes de vectores R son: `NumericVector`, `IntegerVector`, `CharacterVector` y `LogicalVector`.
- Los escalares y los vectores son diferentes. Los equivalentes escalares de vectores numéricos, enteros, de caracteres y lógicos son: `doble`, `int`, `String` y `bool`.
- Debe usar una declaración `return` explícita para devolver un valor de una función.
- Cada instrucción termina con un `;`.

25.2.2. Entrada escalar, salida escalar

La siguiente función de ejemplo implementa una versión escalar de la función `sign()` que devuelve 1 si la entrada es positiva y -1 si es negativa:

```
signR <- function(x) {
  if (x > 0) {
    1
  } else if (x == 0) {
    0
  } else {
    -1
  }
}

cppFunction('int signC(int x) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return -1;
  }
}')
```

En la versión C++:

- Declaramos el tipo de cada entrada de la misma forma que declaramos el tipo de la salida. Si bien esto hace que el código sea un poco más detallado, también aclara el tipo de entrada que necesita la función.
- La sintaxis `if` es idéntica — si bien existen grandes diferencias entre R y C++, ¡también hay muchas similitudes! C++ también tiene

una instrucción `while` que funciona de la misma manera que la de R. Como en R, puede usar `break` para salir del ciclo, pero para omitir una iteración necesita usar `continue` en lugar de `next`.

25.2.3. Entrada vectorial, salida escalar

Una gran diferencia entre R y C++ es que el costo de los bucles es mucho menor en C++. Por ejemplo, podríamos implementar la función `sum` en R usando un bucle. Si ha estado programando en R por un tiempo, ¡probablemente tendrá una reacción visceral a esta función!

```
sumR <- function(x) {
  total <- 0
  for (i in seq_along(x)) {
    total <- total + x[i]
  }
  total
}
```

En C++, los bucles tienen muy poca sobrecarga, por lo que está bien usarlos. En la Section 25.5, verá alternativas a los bucles `for` que expresan más claramente su intención; no son más rápidos, pero pueden hacer que su código sea más fácil de entender.

```
cppFunction('double sumC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total;
}')
```

La versión C++ es similar, pero:

- Para encontrar la longitud del vector, usamos el método `.size()`, que devuelve un número entero. Los métodos de C++ se llaman con `.` (es decir, un punto).
- La sentencia `for` tiene una sintaxis diferente: `for(init; check; increment)`. Este bucle se inicializa creando una nueva variable llamada `i` con valor 0. Antes de cada iteración comprobamos que `i < n`, y terminamos el bucle si no es así. Después de cada iteración, incrementamos el valor de `i` en uno, utilizando el operador de prefijo especial `++` que aumenta el valor de `i` en 1.
- En C++, los índices vectoriales comienzan en 0, lo que significa que el último elemento está en la posición `n - 1`. Repetiré esto porque es muy importante: **¡EN C++, LOS ÍNDICES VECTORIALES EMPIEZAN EN 0!** Esta es una fuente muy común de errores al convertir funciones de R a C++.
- Utilice `=` para la asignación, no `<-`.
- C++ proporciona operadores que modifican en el lugar: `total += x[i]` es equivalente a `total = total + x[i]`. Operadores in situ similares son `-=`, `*=` y `/=`.

Este es un buen ejemplo de dónde C++ es mucho más eficiente que R. Como se muestra en el siguiente micropunto de referencia, `sumC()` es competitivo con el integrado (y altamente optimizado) `sum()`, mientras que `sumR()` es varios órdenes de magnitud más lento.

```
x <- runif(1e3)
bench::mark(
  sum(x),
  sumC(x),
  sumR(x)
)[1:6]
```

```
#> # A tibble: 3 × 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 sum(x)       2.06µs   2.09µs   468078.        0B      0
#> 2 sumC(x)      1.67µs   1.74µs   534937.        0B     53.5
#> 3 sumR(x)     20.09µs  20.56µs    48023.      22KB      0
```

25.2.4. Entrada vectorial, salida vectorial

A continuación, crearemos una función que calcule la distancia euclidiana entre un valor y un vector de valores:

```
pdistR <- function(x, ys) {
  sqrt((x - ys) ^ 2)
}
```

En R, no es obvio que queremos que `x` sea un escalar de la definición de la función, y debemos dejarlo claro en la documentación. Eso no es un problema en la versión de C++ porque tenemos que ser explícitos con los tipos:

```
cppFunction('NumericVector pdistC(double x, NumericVector ys) {
  int n = ys.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  return out;
}')
```

Esta función introduce solo algunos conceptos nuevos:

- Creamos un nuevo vector numérico de longitud `n` con un constructor: `NumericVector out(n)`. Otra forma útil de hacer un vector es copiar uno existente: `NumericVector zs = clone(ys)`.
- C++ usa `pow()`, no `^`, para la exponenciación.

Tenga en cuenta que debido a que la versión R está completamente vectorizada, ya será rápida.

```
y <- runif(1e6)
bench::mark(
  pdistR(0.5, y),
  pdistC(0.5, y)
)[1:6]
#> # A tibble: 2 × 6
#>   expression          min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>     <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 pdistR(0.5, y)   4.92ms   5.27ms      192.    7.63MB     95.8
#> 2 pdistC(0.5, y)   3.99ms   4.09ms      244.    7.63MB    126.
```

En mi computadora, toma alrededor de 5 ms con un vector `y` de 1 millón de elementos. La función de C++ es unas 2,5 veces más rápida, ~2 ms, pero suponiendo que le haya llevado 10 minutos escribir la función de C++, necesitará ejecutarla ~200.000 veces para que valga la pena reescribirla. La razón por la que la función de C++ es más rápida es sutil y se relaciona con la administración de la memoria. La versión R necesita crear un vector intermedio de la misma longitud que y (`x - ys`), y asignar memoria es una operación costosa. La función de C++ evita esta sobrecarga porque usa un escalar intermedio.

25.2.5. Usar sourceCpp

Hasta ahora, hemos usado C++ en línea con `cppFunction()`. Esto hace que la presentación sea más simple, pero para problemas reales, general-

mente es más fácil usar archivos independientes de C++ y luego enviarlos a R usando `sourceCpp()`. Esto le permite aprovechar la compatibilidad con el editor de texto para archivos C++ (p. ej., resaltado de sintaxis), además de facilitar la identificación de los números de línea en los errores de compilación.

Su archivo independiente de C++ debe tener la extensión `.cpp` y debe comenzar con:

```
#include <Rcpp.h>
using namespace Rcpp;
```

Y para cada función que desee que esté disponible dentro de R, debe prefijarla con:

```
// [[Rcpp::export]]
```

Si está familiarizado con roxygen2, puede preguntarse cómo se relaciona esto con `@export`. `Rcpp::export` controla si una función se exporta de C++ a R; `@export` controla si una función se exporta desde un paquete y se pone a disposición del usuario.

Puede incrustar código R en bloques de comentarios especiales de C++. Esto es realmente conveniente si desea ejecutar algún código de prueba:

```
/*** R
# This is R code
*/
```

El código R se ejecuta con `source(echo = TRUE)`, por lo que no es necesario que imprima la salida explícitamente.

Para compilar el código C++, usa `sourceCpp("ruta/al/archivo.cpp")`. Esto creará las funciones R coincidentes y las agregará a su sesión actual.

Tenga en cuenta que estas funciones no se pueden guardar en un archivo `.Rdata` y volver a cargar en una sesión posterior; deben volver a crearse cada vez que reinicie R.

Por ejemplo, ejecutar `sourceCpp()` en el siguiente archivo implementa mean en C++ y luego lo compara con `mean()` integrado:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double meanC(NumericVector x) {
  int n = x.size();
  double total = 0;

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

/*** R
x <- runif(1e5)
bench::mark(
  mean(x),
  meanC(x)
)
*/
```

NB: Si ejecuta este código, notará que `meanC()` es mucho más rápido que el `mean()` integrado. Esto se debe a que cambia la precisión numérica por la velocidad.

En el resto de este capítulo, el código C++ se presentará de forma independiente en lugar de envuelto en una llamada a `cppFunction`. Si desea

intentar compilar y/o modificar los ejemplos, debe pegarlos en un archivo fuente de C++ que incluya los elementos descritos anteriormente. Esto es fácil de hacer en RMarkdown: todo lo que necesita hacer es especificar `engine = "Rcpp"`.

25.2.6. Ejercicios

1. Con los conceptos básicos de C++ en la mano, ahora es un buen momento para practicar leyendo y escribiendo algunas funciones simples de C++. Para cada una de las siguientes funciones, lea el código y averigüe cuál es la función base R correspondiente. Es posible que aún no comprenda cada parte del código, pero debería poder descubrir los conceptos básicos de lo que hace la función.

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

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

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

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

```
}

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

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

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

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

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

  NumericVector out(n);

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

  return out;
```

```
}
```

2. Para practicar sus habilidades de escritura de funciones, convierta las siguientes funciones a C++. Por ahora, suponga que las entradas no tienen valores faltantes.

    1. `all()`.
    2. `cumprod()`, `cummin()`, `cummax()`.
    3. `diff()`. Start by assuming lag 1, and then generalise for lag `n`.
    4. `range()`.
    5. `var()`. Lea acerca de los enfoques que puede adoptar en Wikipedia. Siempre que se implemente un algoritmo numérico, siempre es bueno verificar lo que ya se sabe sobre el problema.

25.3. Otras clases

Ya has visto las clases vectoriales básicas (`IntegerVector`, `NumericVector`, `LogicalVector`, `CharacterVector`) y sus equivalentes escalares (`int`, `double`, `bool`, `String`). Rcpp también proporciona contenedores para todos los demás tipos de datos básicos. Los más importantes son para listas y marcos de datos, funciones y atributos, como se describe a continuación. Rcpp también proporciona clases para más tipos como `Environment`, `DottedPair`, `Language`, `Symbol`, etc., pero estos están más allá del alcance de este capítulo.

25.3.1. Listas y data frames

Rcpp también proporciona las clases `List` y `DataFrame`, pero son más útiles para la salida que para la entrada. Esto se debe a que las listas y los marcos de datos pueden contener clases arbitrarias, pero C++ necesita

conocer sus clases de antemano. Si la lista tiene una estructura conocida (p. ej., es un objeto de S3), puede extraer los componentes y convertirlos manualmente a sus equivalentes de C++ con `as()`. Por ejemplo, el objeto creado por `lm()`, la función que ajusta un modelo lineal, es una lista cuyos componentes son siempre del mismo tipo. El siguiente código ilustra cómo puede extraer el error porcentual medio (`mpe()`) de un modelo lineal. Este no es un buen ejemplo de cuándo usar C++, porque se implementa muy fácilmente en R, pero muestra cómo trabajar con una clase S3 importante. Tenga en cuenta el uso de `.inherits()` y `stop()` para verificar que el objeto realmente es un modelo lineal.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double mpe(List mod) {
  if (!mod.inherits("lm")) stop("Input must be a linear model");

  NumericVector resid = as<NumericVector>(mod["residuals"]);
  NumericVector fitted = as<NumericVector>(mod["fitted.values"]);

  int n = resid.size();
  double err = 0;
  for(int i = 0; i < n; ++i) {
    err += resid[i] / (fitted[i] + resid[i]);
  }
  return err / n;
}
```

```
mod <- lm(mpg ~ wt, data = mtcars)
mpe(mod)
#> [1] -0.0154
```

25.3.2. Funciones

Puede poner funciones R en un objeto de tipo `Function`. Esto hace que llamar a una función R desde C++ sea sencillo. El único desafío es que no sabemos qué tipo de salida devolverá la función, por lo que usamos el tipo general `RObject`.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
RObject callWithOne(Function f) {
  return f(1);
}
```

```
callWithOne(function(x) x + 1)
#> [1] 2
callWithOne(paste)
#> [1] "1"
```

Llamar funciones R con argumentos posicionales es obvio:

```
f("y", 1);
```

Pero necesita una sintaxis especial para argumentos con nombre:

```
f(_["x"] = "y", _["value"] = 1);
```

25.3.3. Atributos

Todos los objetos R tienen atributos, que se pueden consultar y modificar con `.attr()`. Rcpp también proporciona `.names()` como un alias para

el atributo de nombre. El siguiente fragmento de código ilustra estos métodos. Tenga en cuenta el uso de `::create()`, un método de *clase*. Esto le permite crear un vector R a partir de valores escalares de C++:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector attribs() {
  NumericVector out = NumericVector::create(1, 2, 3);

  out.names() = CharacterVector::create("a", "b", "c");
  out.attr("my-attr") = "my-value";
  out.attr("class") = "my-class";

  return out;
}
```

Para los objetos S4, `.slot()` juega un papel similar a `.attr()`.

25.4. Valores faltantes

Si está trabajando con valores perdidos, necesita saber dos cosas:

- Cómo se comportan los valores faltantes de R en los escalares de C++ (por ejemplo, `doble`).
- Cómo obtener y establecer valores faltantes en vectores (por ejemplo, `NumericVector`).

25.4.1. Escalares

El siguiente código explora lo que sucede cuando tomas uno de los valores faltantes de R, lo conviertes en un escalar y luego vuelves a convertirlo en un vector R. Tenga en cuenta que este tipo de experimentación es una forma útil de descubrir qué hace cualquier operación.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List scalar_missings() {
  int int_s = NA_INTEGER;
  String chr_s = NA_STRING;
  bool lgl_s = NA_LOGICAL;
  double num_s = NA_REAL;

  return List::create(int_s, chr_s, lgl_s, num_s);
}
```

```
str(scalar_missings())
#> List of 4
#>  $ : int NA
#>  $ : chr NA
#>  $ : logi TRUE
#>  $ : num NA
```

Con la excepción de `bool`, las cosas se ven bastante bien aquí: todos los valores que faltan se han conservado. Sin embargo, como veremos en las siguientes secciones, las cosas no son tan sencillas como parecen.

25.4.1.1. Enteros

Con números enteros, los valores faltantes se almacenan como el número entero más pequeño. Si no les haces nada, se conservarán. Pero, dado que C++ no sabe que el entero más pequeño tiene este comportamiento especial, si hace algo al respecto, es probable que obtenga un valor incorrecto: por ejemplo, `evalCpp('NA_INTEGER + 1')` da -2147483647.

Entonces, si desea trabajar con valores faltantes en números enteros, use un `IntegerVector` de longitud 1 o tenga mucho cuidado con su código.

25.4.1.2. Dobles

Con los dobles, es posible que pueda ignorar los valores faltantes y trabajar con NaN (no un número). Esto se debe a que el NA de R es un tipo especial de número NaN de punto flotante IEEE 754. Entonces, cualquier expresión lógica que involucre un NaN (o en C++, NAN) siempre se evalúa como FALSE:

```
evalCpp("NAN == 1")
#> [1] FALSE
evalCpp("NAN < 1")
#> [1] FALSE
evalCpp("NAN > 1")
#> [1] FALSE
evalCpp("NAN == NAN")
#> [1] FALSE
```

(Aquí estoy usando `evalCpp()` que le permite ver el resultado de ejecutar una sola expresión de C++, lo que la hace excelente para este tipo de experimentación interactiva.)

Pero tenga cuidado al combinarlos con valores booleanos:

```
evalCpp("NAN && TRUE")
#> [1] TRUE
evalCpp("NAN || FALSE")
#> [1] TRUE
```

Sin embargo, en contextos numéricos, los NaN propagarán los NA:

```
evalCpp("NAN + 1")
#> [1] NaN
evalCpp("NAN - 1")
#> [1] NaN
evalCpp("NAN / 1")
#> [1] NaN
evalCpp("NAN * 1")
#> [1] NaN
```

25.4.2. Caracteres

`String` es una clase de cadena de caracteres escalar introducida por Rcpp, por lo que sabe cómo lidiar con los valores faltantes.

25.4.3. Booleano

Mientras que `bool` de C++ tiene dos valores posibles (`true` o `false`), un vector lógico en R tiene tres (`TRUE`, `FALSE` y `NA`). Si fuerza un vector lógico de longitud 1, asegúrese de que no contenga ningún valor faltante; de lo contrario, se convertirán en VERDADERO. Una solución fácil es usar `int` en su lugar, ya que esto puede representar `TRUE`, `FALSE` y `NA`.

25.4.4. Vectores

Con los vectores, debe usar un valor perdido específico para el tipo de vector, `NA_REAL`, `NA_INTEGER`, `NA_LOGICAL`, `NA_STRING`:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List missing_sampler() {
  return List::create(
    NumericVector::create(NA_REAL),
    IntegerVector::create(NA_INTEGER),
    LogicalVector::create(NA_LOGICAL),
    CharacterVector::create(NA_STRING)
  );
}
```

```
str(missing_sampler())
#> List of 4
#>  $ : num NA
#>  $ : int NA
#>  $ : logi NA
#>  $ : chr NA
```

25.4.5. Ejercicios

1. Vuelva a escribir cualquiera de las funciones del primer ejercicio de la Section 25.2.6 para tratar con los valores faltantes. Si `na.rm` es verdadero, ignore los valores faltantes. Si `na.rm` es falso, devuelve un valor faltante si la entrada contiene valores faltantes. Algunas buenas funciones para practicar son `min()`, `max()`, `range()`, `mean()` y `var()`.

2. Vuelva a escribir `cumsum()` y `diff()` para que puedan manejar los valores faltantes. Tenga en cuenta que estas funciones tienen un comportamiento un poco más complicado.

25.5. Biblioteca de plantillas estándar

La verdadera fortaleza de C++ se revela cuando necesita implementar algoritmos más complejos. La biblioteca de plantillas estándar (STL) proporciona un conjunto de estructuras de datos y algoritmos extremadamente útiles. Esta sección explicará algunos de los algoritmos y estructuras de datos más importantes y le indicará la dirección correcta para obtener más información. No puedo enseñarte todo lo que necesitas saber sobre STL, pero espero que los ejemplos te muestren el poder de STL y te convenzan de que es útil aprender más.

Si necesita un algoritmo o una estructura de datos que no esté implementado en STL, un buen lugar para buscar es boost. La instalación de boost en su computadora está más allá del alcance de este capítulo, pero una vez que lo haya instalado, puede usar las estructuras de datos y los algoritmos de boost al incluir el archivo de encabezado apropiado con (p. ej.) `#include <boost/array.hpp>`.

25.5.1. Usar iteradores

Los iteradores se utilizan mucho en STL: muchas funciones aceptan o devuelven iteradores. Son el siguiente paso de los bucles básicos, abstrayendo los detalles de la estructura de datos subyacente. Los iteradores tienen tres operadores principales:

1. Avanza con `++`.
2. Obtener el valor al que se refieren, o **desreferenciar**, con `*`.
3. Comparar con `==`.

Por ejemplo, podríamos reescribir nuestra función de suma usando iteradores:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum3(NumericVector x) {
  double total = 0;

  NumericVector::iterator it;
  for(it = x.begin(); it != x.end(); ++it) {
    total += *it;
  }
  return total;
}
```

Los principales cambios están en el bucle for:

- Comenzamos en `x.begin()` y repetimos hasta llegar a `x.end()`. Una pequeña optimización es almacenar el valor del iterador final para que no tengamos que buscarlo cada vez. Esto solo ahorra alrededor de 2 ns por iteración, por lo que solo es importante cuando los cálculos en el ciclo son muy simples.
- En lugar de indexar en x, usamos el operador de desreferencia para obtener su valor actual: `*it`.
- Observe el tipo de iterador: `NumericVector::iterator`. Cada tipo de vector tiene su propio tipo de iterador: `LogicalVector::iterator`, `CharacterVector::iterator`, etc.

Este código se puede simplificar aún más mediante el uso de una característica de C++11: bucles for basados en rangos. C ++ 11 está ampliamente

disponible y se puede activar fácilmente para usar con Rcpp agregando `[[Rcpp::plugins(cpp11)]]`.

```
// [[Rcpp::plugins(cpp11)]]
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum4(NumericVector xs) {
  double total = 0;

  for(const auto &x : xs) {
    total += x;
  }
  return total;
}
```

Los iteradores también nos permiten usar los equivalentes de C++ de la familia de funciones apply. Por ejemplo, podríamos volver a escribir `sum()` para usar la función `accumulate()`, que toma un iterador inicial y final, y suma todos los valores en el vector. El tercer argumento para `acumular` da el valor inicial: es particularmente importante porque esto también determina el tipo de datos que usa `acumular` (así que usamos `0.0` y no `0` para que `acumule` use un `doble`, no un `int`.). Para usar `accumulate()` necesitamos incluir el encabezado `<numeric>`.

```
#include <numeric>
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum5(NumericVector x) {
  return std::accumulate(x.begin(), x.end(), 0.0);
}
```

25.5.2. Algoritmos

El encabezado `<algorithm>` proporciona una gran cantidad de algoritmos que funcionan con iteradores. Una buena referencia está disponible en https://en.cppreference.com/w/cpp/algorithm. Por ejemplo, podríamos escribir una versión básica de Rcpp de `findInterval()` que toma dos argumentos, un vector de valores y un vector de rupturas, y ubica el contenedor en el que cae cada x. Esto muestra algunas características más avanzadas del iterador. Lea el código a continuación y vea si puede descubrir cómo funciona.

```
#include <algorithm>
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector findInterval2(NumericVector x, NumericVector breaks) {
  IntegerVector out(x.size());

  NumericVector::iterator it, pos;
  IntegerVector::iterator out_it;

  for(it = x.begin(), out_it = out.begin(); it != x.end();
      ++it, ++out_it) {
    pos = std::upper_bound(breaks.begin(), breaks.end(), *it);
    *out_it = std::distance(breaks.begin(), pos);
  }

  return out;
}
```

Los puntos clave son:

- Pasamos por dos iteradores (entrada y salida) simultáneamente.

- Podemos asignar a un iterador desreferenciado (`out_it`) para cambiar los valores en `out`.
- `upper_bound()` devuelve un iterador. Si quisiéramos el valor de `upper_bound()`, podríamos quitarle la referencia; para averiguar su ubicación, usamos la función `distance()`.
- Pequeña nota: si queremos que esta función sea tan rápida como `findInterval()` en R (que usa código C escrito a mano), necesitamos calcular las llamadas a `.begin()` y `.end()` una vez y guardar los resultados. Esto es fácil, pero distrae la atención de este ejemplo, por lo que se ha omitido. Hacer este cambio produce una función que es un poco más rápida que la función `findInterval()` de R, pero es aproximadamente 1/10 del código.

En general, es mejor usar algoritmos de STL que bucles enrollados a mano. En *Effective STL*, Scott Meyers da tres razones: eficiencia, corrección y mantenibilidad. Los algoritmos de STL están escritos por expertos en C++ para que sean extremadamente eficientes y han existido durante mucho tiempo, por lo que están bien probados. El uso de algoritmos estándar también hace que la intención de su código sea más clara, lo que ayuda a que sea más legible y fácil de mantener.

25.5.3. Estructuras de datos

El STL proporciona un gran conjunto de estructuras de datos: `array`, `bitset`, `list`, `forward_list`, `map`, `multimap`, `multiset`, `priority_queue`, `queue`, `deque`, `set`, `stack`, `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`, y `vector`. Las más importantes de estas estructuras de datos son el `vector`, el `unordered_set` y el `unordered_map`. Nos centraremos en estos tres en esta sección, pero el uso de los otros es similar: solo tienen diferentes compensaciones de rendimiento. Por ejemplo, `deque` (pronunciado "deck") tiene una interfaz muy similar a los vectores pero una implementación subyacente diferente

que tiene diferentes compensaciones de rendimiento. Es posible que desee probarlo para su problema. Una buena referencia para las estructuras de datos STL es https://en.cppreference.com/w/cpp/container — Le recomiendo que lo mantenga abierto mientras trabaja con STL.

Rcpp sabe cómo convertir muchas estructuras de datos STL a sus equivalentes R, por lo que puede devolverlas desde sus funciones sin convertirlas explícitamente en estructuras de datos R.

25.5.4. Vectores

Un vector STL es muy similar a un vector R, excepto que crece de manera eficiente. Esto hace que los vectores sean apropiados para usar cuando no se sabe de antemano qué tan grande será la salida. Los vectores tienen una plantilla, lo que significa que debe especificar el tipo de objeto que contendrá el vector cuando lo cree: `vector<int>`, `vector<bool>`, `vector<double>`, `vector<String>`. Puede acceder a elementos individuales de un vector usando la notación estándar `[]`, y puede agregar un nuevo elemento al final del vector usando `.push_back()`. Si tiene una idea de antemano de qué tan grande será el vector, puede usar `.reserve()` para asignar suficiente almacenamiento.

El siguiente código implementa la codificación de longitud de ejecución (`rle()`). Produce dos vectores de salida: un vector de valores y un vector de "longitudes" que indica cuántas veces se repite cada elemento. Funciona recorriendo el vector de entrada `x` comparando cada valor con el anterior: si es el mismo, incrementa el último valor en `lengths`; si es diferente, agrega el valor al final de `values` y establece la longitud correspondiente en 1.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
```

```
List rleC(NumericVector x) {
  std::vector<int> lengths;
  std::vector<double> values;

  // Initialise first value
  int i = 0;
  double prev = x[0];
  values.push_back(prev);
  lengths.push_back(1);

  NumericVector::iterator it;
  for(it = x.begin() + 1; it != x.end(); ++it) {
    if (prev == *it) {
      lengths[i]++;
    } else {
      values.push_back(*it);
      lengths.push_back(1);

      i++;
      prev = *it;
    }
  }

  return List::create(
    _["lengths"] = lengths,
    _["values"] = values
  );
}
```

(Una implementación alternativa sería reemplazar `i` con el iterador `lengths.rbegin()` que siempre apunta al último elemento del vector. Es posible que desee intentar implementar eso.)

Otros métodos de un vector se describen en https://en.cppreference.com/

w/cpp/container/vector.

25.5.5. Conjuntos

Los conjuntos mantienen un conjunto único de valores y pueden indicar de manera eficiente si ha visto un valor antes. Son útiles para problemas que involucran duplicados o valores únicos (como `unique`, `duplicated` o `in`). C++ proporciona tanto conjuntos ordenados (`std::set`) como desordenados (`std::unordered_set`), dependiendo de si el orden es importante para usted o no. Los conjuntos desordenados tienden a ser mucho más rápidos (porque usan una tabla hash internamente en lugar de un árbol), por lo que incluso si necesita un conjunto ordenado, debe considerar usar un conjunto desordenado y luego ordenar la salida. Al igual que los vectores, los conjuntos tienen plantillas, por lo que debe solicitar el tipo de conjunto adecuado para su propósito: `unordered_set<int>`, `unordered_set<bool>`, etc. Hay más detalles disponibles en https://en.cppreference.com/w/cpp/container/set y https://en.cppreference.com/w/cpp/container/unordered_set.

La siguiente función usa un conjunto desordenado para implementar un equivalente a `duplicated()` para vectores enteros. Tenga en cuenta el uso de `seen.insert(x[i]).second`. `insert()` devuelve un par, el valor `.first` es un iterador que apunta al elemento y el valor `.second` es un valor booleano que es verdadero si el valor fue una nueva adición al conjunto.

```
// [[Rcpp::plugins(cpp11)]]
#include <Rcpp.h>
#include <unordered_set>
using namespace Rcpp;

// [[Rcpp::export]]
LogicalVector duplicatedC(IntegerVector x) {
  std::unordered_set<int> seen;
```

```
  int n = x.size();
  LogicalVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = !seen.insert(x[i]).second;
  }

  return out;
}
```

25.5.6. Map

Un mapa es similar a un conjunto, pero en lugar de almacenar presencia o ausencia, puede almacenar datos adicionales. Es útil para funciones como `table()` o `match()` que necesitan buscar un valor. Al igual que con los conjuntos, existen versiones ordenadas (`std::map`) y desordenadas (`std::unordered_map`). Dado que los mapas tienen un valor y una clave, debe especificar ambos tipos al inicializar un mapa: `mapa<doble, int>`, `mapa_desordenado<int, doble>`, etc. El siguiente ejemplo muestra cómo podrías usar un `map` para implementar `table()` para vectores numéricos:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
std::map<double, int> tableC(NumericVector x) {
  std::map<double, int> counts;

  int n = x.size();
  for (int i = 0; i < n; i++) {
    counts[x[i]]++;
  }
```

```
  return counts;
}
```

25.5.7. Ejercicios

Para practicar el uso de estructuras de datos y algoritmos STL, implemente lo siguiente mediante funciones R en C++, utilizando las sugerencias proporcionadas:

1. `median.default()` usando `partial_sort`.
2. `%in%` usando `unordered_set` y los `find()` o `count()` métodos.
3. `unique()` usando un `unordered_set` (desafío: ¡hazlo en una línea!).
4. `min()` usando `std::min()`, o `max()` usando `std::max()`.
5. `which.min()` usando `min_element`, o `which.max()` usando `max_element`.
6. `setdiff()`, `union()`, y `intersect()` para números enteros usando rangos ordenados y `set_union`, `set_intersection` y `set_difference`.

25.6. Caso de estudio

Los siguientes estudios de casos ilustran algunos usos reales de C++ para reemplazar el código R lento.

25.6.1. Muestreador de Gibbs

El siguiente estudio de caso actualiza un ejemplo sobre el que se escribió en un blog de Dirk Eddelbuettel, que ilustra la conversión de una muestra de Gibbs en R a C++. El código R y C++ que se muestra a continuación es muy similar (solo tomó unos minutos convertir la versión R a la versión C++), pero se ejecuta unas 20 veces más rápido en mi computadora. La publicación del blog de Dirk también muestra otra forma de hacerlo aún más rápido: usar las funciones de generador de números aleatorios más rápidas en GSL (fácilmente accesible desde R a través del paquete RcppGSL) puede hacerlo otras dos o tres veces más rápido.

El código R es el siguiente:

```
gibbs_r <- function(N, thin) {
  mat <- matrix(nrow = N, ncol = 2)
  x <- y <- 0

  for (i in 1:N) {
    for (j in 1:thin) {
      x <- rgamma(1, 3, y * y + 4)
      y <- rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))
    }
    mat[i, ] <- c(x, y)
  }
  mat
}
```

Esto es fácil de convertir a C++. Nosotros:

- Agregar declaraciones de tipo a todas las variables.
- Utilice `(` en lugar de `[` para indexar en la matriz.

- Subíndice los resultados de `rgamma` y `rnorm` para convertir de un vector a un escalar.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix gibbs_cpp(int N, int thin) {
  NumericMatrix mat(N, 2);
  double x = 0, y = 0;

  for(int i = 0; i < N; i++) {
    for(int j = 0; j < thin; j++) {
      x = rgamma(1, 3, 1 / (y * y + 4))[0];
      y = rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))[0];
    }
    mat(i, 0) = x;
    mat(i, 1) = y;
  }

  return(mat);
}
```

La evaluación comparativa de los dos rendimientos de implementación:

```
bench::mark(
  gibbs_r(100, 10),
  gibbs_cpp(100, 10),
  check = FALSE
)
#> # A tibble: 2 × 6
#>   expression             min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>         <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
```

```
#> 1 gibbs_r(100, 10)     2.44ms    2.5ms      398.    107.5KB     12.8
#> 2 gibbs_cpp(100, 10) 243.74µs  260.8µs     3789.     1.61KB     16.7
```

25.6.2. R vectorización frente a vectorización C++

Este ejemplo está adaptado de "Rcpp fuma rápido para modelos basados en agentes en marcos de datos". El desafío es predecir la respuesta de un modelo a partir de tres entradas. La versión básica de R del predictor se ve así:

```
vacc1a <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * if (female) 1.25 else 0.75
  p <- max(0, p)
  p <- min(1, p)
  p
}
```

Queremos poder aplicar esta función a muchas entradas, por lo que podríamos escribir una versión de entrada vectorial usando un bucle for.

```
vacc1 <- function(age, female, ily) {
  n <- length(age)
  out <- numeric(n)
  for (i in seq_len(n)) {
    out[i] <- vacc1a(age[i], female[i], ily[i])
  }
  out
}
```

Si está familiarizado con R, tendrá el presentimiento de que esto será lento, y de hecho lo es. Hay dos formas en las que podemos atacar este

problema. Si tiene un buen vocabulario de R, puede ver inmediatamente cómo vectorizar la función (usando `ifelse()`, `pmin()` y `pmax()`). Alternativamente, podríamos reescribir `vacc1a()` y `vacc1()` en C++, utilizando nuestro conocimiento de que los bucles y las llamadas a funciones tienen una sobrecarga mucho menor en C++.

Cualquiera de los dos enfoques es bastante sencillo. En R:

```
vacc2 <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * ifelse(female, 1.25, 0.75)
  p <- pmax(0, p)
  p <- pmin(1, p)
  p
}
```

(Si ha trabajado mucho con R, es posible que reconozca algunos cuellos de botella potenciales en este código: se sabe que `ifelse`, `pmin` y `pmax` son lentos y podrían reemplazarse con `p * 0.75 + p * 0.5 * mujer`, `p[p < 0] <- 0`, `p[p > 1] <- 1`. Es posible que desee intentar cronometrar esas variaciones.)

O en C++:

```
#include <Rcpp.h>
using namespace Rcpp;

double vacc3a(double age, bool female, bool ily){
  double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
  p = p * (female ? 1.25 : 0.75);
  p = std::max(p, 0.0);
  p = std::min(p, 1.0);
  return p;
}
```

```
// [[Rcpp::export]]
NumericVector vacc3(NumericVector age, LogicalVector female,
                    LogicalVector ily) {
  int n = age.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = vacc3a(age[i], female[i], ily[i]);
  }

  return out;
}
```

A continuación, generamos algunos datos de muestra y verificamos que las tres versiones devuelvan los mismos valores:

```
n <- 1000
age <- rnorm(n, mean = 50, sd = 10)
female <- sample(c(T, F), n, rep = TRUE)
ily <- sample(c(T, F), n, prob = c(0.8, 0.2), rep = TRUE)

stopifnot(
  all.equal(vacc1(age, female, ily), vacc2(age, female, ily)),
  all.equal(vacc1(age, female, ily), vacc3(age, female, ily))
)
```

La publicación del blog original olvidó hacer esto e introdujo un error en la versión C++: usaba `0.004` en lugar de `0.04`. Finalmente, podemos comparar nuestros tres enfoques:

```
bench::mark(
  vacc1 = vacc1(age, female, ily),
```

```
  vacc2 = vacc2(age, female, ily),
  vacc3 = vacc3(age, female, ily)
)
#> # A tibble: 3 × 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 vacc1        1.51ms   1.54ms      645.    7.86KB     19.2
#> 2 vacc2        42.1µs   44.1µs    21489.  148.56KB     45.0
#> 3 vacc3       11.33µs  11.61µs    85301.   14.48KB     17.1
```

No es sorprendente que nuestro enfoque original con bucles sea muy lento. La vectorización en R proporciona una gran aceleración, y podemos obtener aún más rendimiento (unas diez veces) con el ciclo de C++. Me sorprendió un poco que C++ fuera mucho más rápido, pero se debe a que la versión R tiene que crear 11 vectores para almacenar resultados intermedios, mientras que el código C++ solo necesita crear 1.

25.7. Using Rcpp in a package

El mismo código C++ que se usa con `sourceCpp()` también se puede agrupar en un paquete. Hay varios beneficios de mover el código de un archivo fuente de C++ independiente a un paquete:

1. Su código puede estar disponible para los usuarios sin las herramientas de desarrollo de C++.

2. El sistema de compilación de paquetes R gestiona automáticamente varios archivos de origen y sus dependencias.

3. Los paquetes brindan infraestructura adicional para pruebas, documentación y consistencia.

Para agregar `Rcpp` a un paquete existente, coloque sus archivos C++ en el directorio `src/` y cree o modifique los siguientes archivos de configuración:

- En `DESCRIPTION` añade

  ```
  LinkingTo: Rcpp
  Imports: Rcpp
  ```

- Asegúrese de que su `NAMESPACE` incluye:

  ```
  useDynLib(mypackage)
  importFrom(Rcpp, sourceCpp)
  ```

  Necesitamos importar algo (cualquier cosa) de Rcpp para que el código interno de Rcpp se cargue correctamente. Este es un error en R y, con suerte, se solucionará en el futuro.

La forma más fácil de configurar esto automáticamente es llamar a `usethis::use_rcpp()`.

Antes de compilar el paquete, deberá ejecutar `Rcpp::compileAttributes()`. Esta función escanea los archivos C++ en busca de atributos `Rcpp::export` y genera el código necesario para que las funciones estén disponibles en R. Vuelva a ejecutar `compileAttributes()` cada vez que se agreguen, eliminen o cambien sus firmas. Esto lo hace automáticamente el paquete devtools y Rstudio.

Para obtener más detalles, consulte la viñeta del paquete Rcpp, `vignette("Rcpp-package")`.

25.8. Aprendiendo más

Este capítulo solo ha tocado una pequeña parte de Rcpp, brindándole las herramientas básicas para reescribir código R de bajo rendimiento en

C++. Como se señaló, Rcpp tiene muchas otras capacidades que facilitan la interfaz de R con el código C++ existente, que incluyen:

- Características adicionales de los atributos, incluida la especificación de argumentos predeterminados, la vinculación en dependencias externas de C++ y la exportación de interfaces de C++ desde paquetes. Estas características y más están cubiertas en la viñeta de atributos de Rcpp, `vignette("Rcpp-attributes")`.
- Creación automática de contenedores entre estructuras de datos de C++ y estructuras de datos de R, incluida la asignación de clases de C++ a clases de referencia. Una buena introducción a este tema es la viñeta de módulos Rcpp, `vignette("Rcpp-modules")`.
- La guía de referencia rápida de Rcpp, `vignette("Rcpp-quickref")`, contiene un resumen útil de las clases de Rcpp y lenguajes de programación comunes.

Recomiendo encarecidamente estar atento a la página de inicio de Rcpp y registrarse en la lista de correo de Rcpp.

Otros recursos que he encontrado útiles para aprender C++ son:

- *Effective C++* (Meyers 2005) y *Effective STL* (Meyers 2001).
- *C++ Annotations*, dirigido a usuarios expertos en C (o cualquier otro lenguaje que use una gramática similar a C, como Perl o Java) que deseen saber más sobre C++ o hacer la transición a C++.
- *Algorithm Libraries*, que proporciona una descripción más técnica, pero aún concisa, de conceptos importantes de STL. (Siga los enlaces debajo de las notas).

Escribir código de rendimiento también puede requerir que reconsidere su enfoque básico: una sólida comprensión de las estructuras de datos y los algoritmos básicos es muy útil aquí. Eso está más allá del alcance de este libro, pero sugeriría el *Manual de diseño de algoritmos* (Skiena 1998),

Introducción a los algoritmos del MIT, *Algorithms* de Robert Sedgewick y Kevin Wayne, que tiene un libro de texto en línea gratuito y un [curso de Coursera] correspondiente (https://www.coursera.org/learn/algorithms-part1).

25.9. Reconocimientos

Me gustaría agradecer a la lista de correo de Rcpp por muchas conversaciones útiles, en particular a Romain Francois y Dirk Eddelbuettel, quienes no solo han brindado respuestas detalladas a muchas de mis preguntas, sino que también han sido increíblemente receptivos para mejorar Rcpp. Este capítulo no hubiera sido posible sin JJ Allaire; me animó a aprender C++ y luego respondió muchas de mis preguntas tontas en el camino.

Referencias

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. MIT Press.

Bache, Stefan Milton, and Hadley Wickham. 2014. *Magrittr: A Forward-Pipe Operator for R*. http://magrittr.tidyverse.org/.

Balamuta, James. 2018a. *Errorist: Automatically Search Errors or Warnings*. https://github.com/coatless/errorist.

———. 2018b. *Searcher: Query Search Interfaces*. https://github.com/coatless/searcher.

Bates, Douglas, and Martin Maechler. 2018. "Matrix: Sparse and Dense Matrix Classes and Methods." https://CRAN.R-project.org/package=Matrix.

Bawden, Alan. 1999. "Quasiquotation in Lisp." In *PEPM '99*, 4–12. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.309.227.

Bengtsson, Henrik. 2003. "The R.oo Package - Object-Oriented Programming with References Using Standard R Code." In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, edited by Kurt Hornik, Friedrich Leisch, and Achim Zeileis. Vienna, Austria: https://www.r-project.org/conferences/DSC-2003/Proceedings/. https://www.r-project.org/conferences/DSC-2003/Proceedings/Bengtsson.pdf.

Brown, Christopher. 2013. *Hash: Full Feature Implementation of Hash/Associated Arrays/Dictionaries*. https://CRAN.R-project.org/package=hash.

Bueno, Carlos. 2013. *Mature Optimization Handbook*. http://carlos.bueno.org/optimization/.

Chambers, John M. 1998. *Programming with Data: A Guide to the s Language*. Springer.

———. 2008. *Software for Data Analysis: Programming with R*. Springer.

———. 2014. "Object-Oriented Programming, Functional Programming and R." *Statistical Science* 29 (2): 167–80. https://projecteuclid.org/download/pdfview_1/euclid.ss/1408368569.

———. 2016. *Extending R*. CRC Press.

Chambers, John M, and Trevor J Hastie. 1992. *Statistical Models in S*. Wadsworth & Brooks/Cole Advanced Books & Software.

Chang, Winston. 2017. *R6: Classes with Reference Semantics*. https://r6.r-lib.org.

Eddelbuettel, Dirk, and Romain François. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18. https://doi.org/10.18637/jss.v040.i08.

Fowler, Martin. 2010. *Domain-Specific Languages*. Pearson Education. http://amzn.com/0321712943.

Grolemund, Garrett, and Hadley Wickham. 2011. "Dates and Times Made Easy with lubridate." *Journal of Statistical Software* 40 (3): 1–25. http://www.jstatsoft.org/v40/i03/.

Grothendieck, Gabor, Louis Kates, and Thomas Petzoldt. 2016. *Proto: Prototype Object-Based Programming*. https://CRAN.R-project.org/package=proto.

Henry, Lionel, and Hadley Wickham. 2018a. *Purrr: Functional Programming Tools*. https://purrr.tidyverse.org.

———. 2018b. *Rlang: Tools for Low-Level R Programming*. https://rlang.r-lib.org.

Hester, Jim. 2018. *Bench: High Precision Timing of R Expressions*. http://bench.r-lib.org/.

Hester, Jim, Kirill Müller, Kevin Ushey, Hadley Wickham, and Winston Chang. 2018. *Withr: Run Code with Temporarily Modified Global State*. http://withr.r-lib.org.

Hunt, Andrew, and David Thomas. 1990. *The Pragmatic Programmer*. Addison Wesley.

Lumley, Thomas. 2001. "Programmer's Niche: Macros in R." *R News*

1 (3): 11–13. https://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf.

Matloff, Norman. 2011. *The Art of R Programming*. No Starch Press.

———. 2015. *Parallel Computing for Data Science*. Chapman & Hall/CRC. http://amzn.com/1466587016.

McCallum, Q. Ethan, and Steve Weston. 2011. *Parallel R*. O'Reilly. http://amzn.com/B005Z29QT4.

Meyers, Scott. 2001. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Pearson Education. http://amzn.com/0201749629.

———. 2005. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Pearson Education. http://amzn.com/0321334876.

Morandat, Floréal, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. "Evaluating the Design of the R Language." In *European Conference on Object-Oriented Programming*, 104–31. Springer. http://r.cs.purdue.edu/pub/ecoop12.pdf.

Müller, Kirill, and Lorenz Walthert. 2018. *Styler: Non-Invasive Pretty Printing of R Code*. http://styler.r-lib.org.

Müller, Kirill, and Hadley Wickham. 2018. *Tibble: Simple Data Frames*. http://tibble.tidyverse.org/.

R Core Team. 2018a. "R Internals." *R Foundation for Statistical Computing*. https://cran.r-project.org/doc/manuals/r-devel/R-ints.html.

———. 2018b. "Writing R Extensions." *R Foundation for Statistical Computing*. https://cran.r-project.org/doc/manuals/r-devel/R-exts.html.

Skiena, Steven S. 1998. *The Algorithm Design Manual*. Springer. http://amzn.com/0387948600.

Teetor, Nathan. 2018. *Zeallot: Multiple, Unpacking, and Destructuring Assignment*. https://CRAN.R-project.org/package=zeallot.

Tierney, Luke, and Riad Jarjour. 2016. *Proftools: Profile Output Processing Tools for R*. https://CRAN.R-project.org/package=proftools.

Van-Roy, Peter, and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. MIT press.

Wickham, Hadley. 2011. "Mutatr: Mutable Objects for R." *Computa-*

tional Statistics 26 (3): 405--418. https://doi.org/10.1007/s00180-011-0235-7.

———. 2018. *Forcats: Tools for Working with Categorical Variables*. http://forcats.tidyverse.org.

Wickham, Hadley, Jim Hester, Kirill Müller, and Daniel Cook. 2018. *Memoise: Memoisation of Functions*. https://github.com/r-lib/memoise.

Wickham, Hadley, and Yihui Xie. 2018. *Evaluate: Parsing and Evaluation Tools That Provide More Details Than the Default*. https://github.com/r-lib/evaluate.