ARMONIZACIÓN DE LAS ESTADÍSTICAS PROVINCIALES DE TURISMO

DOCUMENTO TÉCNICO #6

Ciencia de Datos para el Turismo

Dirección Nacional de Mercados y Estadística
SUBSECRETARÍA DE DESARROLLO ESTRATÉGICO

Ministerio de Turismo y Deportes
Argentina

Documento Técnico N°6: Ciencia de Datos para el Turismo

Herramientas Computacionales para el Análisis de Datos

Dirección Nacional de Mercados y Estadísticas - Subsecretaría de Desarrollo Estratégico

25 de November de 2021

Índice general

Presentación

El presente documento, **Ciencia de Datos para Turismo**, se enmarca en el proyecto de Armonización de las Estadísticas de Turismo en las Provincias de la Dirección Nacional de Mercados y Estadística de la Subsecretaría de Desarrollo Estratégico del Ministerio de Turismo y Deportes. El objetivo general de este proyecto es contribuir con propuestas metodológicas para los sistemas de estadísticas de turismo provinciales que orienten a producir indicadores provinciales básicos y comparables.

Además de este, se encuentra disponible una serie de documentos técnicos que abordan otras problemáticas vinculadas a la producción de estadística de turismo:

- Documento Técnico #1: Conceptos y elementos básicos para la medición provincial de los turistas
- Documento Técnico #2: Propuestas metodológicas para las encuestas de ocupación en alojamientos turísticos
- Documento Técnico #3: Descripción, análisis y utilización de los Registros Administrativos para la medición del Turismo
- Documento Técnico #4: Propuestas Metodológicas para las Encuestas de Perfil del Visitante
- Documento Técnico #5: Medición de la contribución económica del turismo: actividad y empleo

Documento Técnico Nº6 - Resumen

La ciencia de datos es una disciplina que ha brindado nuevas y maravillosas posibilidades a muchas industrias por medio de la explotación de datos. Junto con estas posibilidades, también ha traído consigo cambios y desafíos constantes. La industria del turismo no es una excepción.

En este documento técnico realizaremos una introducción al concepto de ciencia de datos y su proceso. Introduciremos el lenguaje de programación R como la

caja de herramientas principales para poder llevar adelante cada tarea y etapa de este proceso.

El documento se divide en 11 capítulos con ejemplos prácticos y ejercicios (desafíos) para introducir y practicar los conceptos mencionados.

Capítulo 1

Ciencia de Datos

> "Disciplina **emergente** que se basa en el conocimiento en **metodología estadística y ciencias de la computación** para crear predicciones, clasificaciones e ideas impactantes para una amplia gama de campos tradicionales"

No existe un acuerdo sobre una definición formal de ciencia de datos, pero la mayoría de estas definiciones concuerda en que tiene al menos tres pilares: el conocimiento estadístico, el conocimiento de ciencias de la computación y el conocimiento de área sobre el cual se va a aplicar. En este caso el turismo.

El proceso de ciencia de datos en el cual nos vamos a basar se puede ver en el siguiente diagrama:

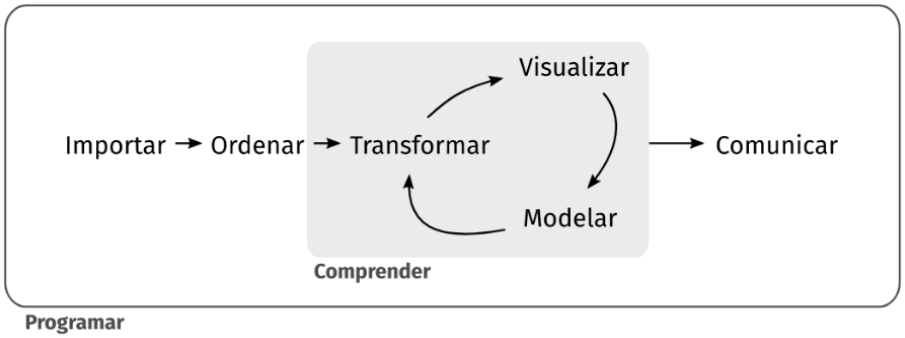


Figura 1.1: Mapa conceptual del proceso de ciencia de datos

Primero, debes **importar** tus datos hacia la herramienta donde vas a procesarlos. Típicamente, esto implica tomar datos que están guardados en un archivo o base de datos y cargarlos en tu software para poder trabajar con ellos.

Una vez que has importado los datos, el siguiente paso es **ordenarlos** para que tengan un formato adecuado para su análisis. Este formato pensado para el análisis tiene la característica que, en los conjuntos de datos ordenados, *cada columna es una variable y cada fila una observación*. Tener datos ordenados nos provee una estructura consistente, preparada para analizarlos y podemos enfocar nuestros esfuerzos en las preguntas que queremos contestar con nuestros datos y no tener que acomodarlos cada vez que la pregunta cambie.

Cuando tus datos están ordenados, podemos necesitar *transformarlos*. La transformación implica quedarte con las observaciones que sean de interés (como todos los hoteles de una ciudad o todos los datos del último año), crear nuevas variables que a partir de variables ya existentes (como calcular el porcentaje de ocupación a partir de la cantidad de plazas totales y las ocupadas) y calcular una serie de estadísticos de resumen (como recuentos y medias).

Una vez que tienes los datos ordenados con las variables que necesitas, hay dos principales fuentes generadoras de conocimiento: la **visualización** y el **modelado**. Ambas tienen fortalezas y debilidades complementarias, por lo que cualquier análisis va a utilizarlas varias veces aprovechando los resultados de una para alimentar a la otra.

La visualización es una herramienta fundamental. Una buena visualización te mostrará el patrón de los datos, cosas que tal vez no esperabas o te hará surgir nuevas preguntas. También puede ayudarte a replantear tus preguntas o darte cuenta si necesitas recolectar datos diferentes.

Los modelos son herramientas complementarias a la visualización. Una vez que tus preguntas son lo suficientemente precisas, puedes utilizar un modelo para responderlas. Los modelos son herramientas estadísticas o computacionales y tienen supuestos para poder aplicarlos, así que la tarea de seleccionar el modelo adecuado para nuestro problema es una parte importante de este proceso, como también lo es su implantación e interpretación posterior.

El último paso en el proceso de la ciencia de datos es la **comunicación**, una parte crítica de cualquier proyecto de análisis de datos, porque es cuando vas a mostrar tus resultados a otras personas y necesitas que puedan comprenderlos y encontrarlos útiles para utilizarlos.

Alrededor de todas estas herramientas se encuentra la **programación** como herramienta transversal en el proyecto de ciencia de datos. No necesitás ser una persona experta en programación para hacer ciencia de datos, pero aprender más sobre programar te ayudará a automatizar tareas recurrentes, compartir tu trabajo de forma reusable y aprovechar el trabajo de otros para resolver problemas similares con mayor facilidad y rapidez.

En este cuadernillo te mostraremos como realizar cada una de estas etapas utilizando el software R y te dejaremos links donde puedes aprender y profundizar más cada aspecto de este proceso.

1.1. ¿Por qué R?

Excel es un software admirable. Es genial para hacer data entry, para ver los datos crudos y para hacer gráficos rápidos. Si venís usándolo hace tiempo, seguro que aprendiste un montón de trucos para sacarle el jugo al máximo, habrás aprendido a usar fórmulas, tablas dinámicas, e incluso macros. Pero seguro que también sufriste sus limitaciones.

En una hoja de Excel no hay un límite claro entre datos y análisis. Sobrescribir datos es un peligro muy real y análisis complicados son imposibles de entender, especialmente si abrís una hoja de cálculo armada por otra persona (que quizás es tu vos del pasado). Además, repetir el análisis en datos distintos o cambiando algún parámetro se puede volver muy engorroso.

Si lo que necesitás son reportes frecuentes y automáticos, y análisis de datos con muchas partes móviles, estaría bueno poder escribir una receta paso a paso y que la computadora corra todo automáticamente cada vez que se lo pedís. Para poder hacer eso, ese paso a paso tiene que estar escrito en un lenguaje que la computadora pueda entender, ese lenguaje es R.

La forma en la que interactuás con la computadora con R es diametralmente distinta que con Excel. Esto lo hace extremadamente poderoso, pero el precio a pagar es básicamente el de tener que aprender un nuevo idioma.

1.2. Cómo decirle a R qué hacer

1.2.1. Orientándose en RStudio

En principio se podría escribir código de R con el Bloc de Notas y luego ejecutarlo, pero nosotros vamos a usar RStudio, que brinda una interfaz gráfica con un montón de herramientas extra para hacernos la vida más fácil.

Cuando abras RStudio te vas a encontrar con una ventana con cuatro paneles como esta:

Los dos paneles de la izquierda son las dos formas principales de interactuar con R. El panel de abajo a la izquierda es **la consola**. Es el lugar que te permite *conversar* con R. Podés escribir comandos que se van a ejecutar inmediátamente cuando aprietes Enter y cuyo resultado se va a mostrar en la consola.

Por ejemplo, hacé click en la consola, escribí `2 + 2` y apretá Enter. Vas a ver algo como esto:

```
2 + 2
```

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

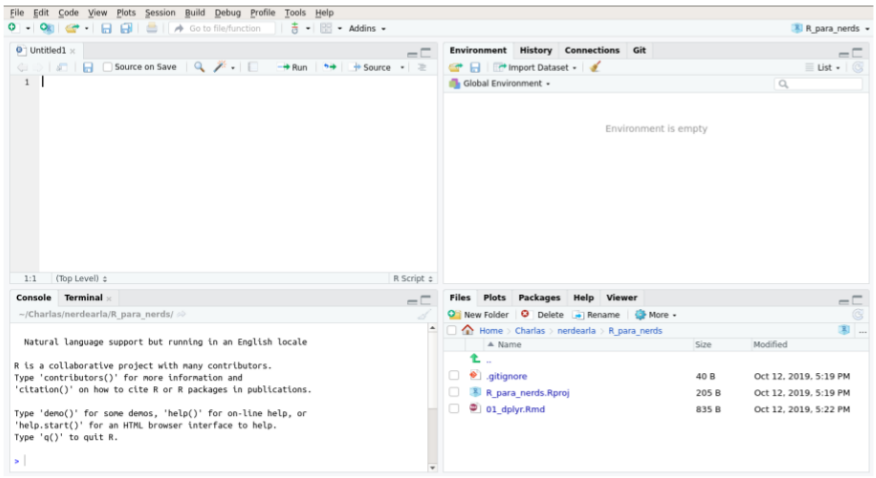


Figura 1.2: Ventana de RStudio

Le dijiste a R que sume 2 y 2 y R te devolvió el resultado: 4 (no te preocupes del `[1]` por ahora). Eso está bueno si querés hacer una cuenta rápida o chequear algo pequeño, pero no sirve para hacer un análisis complejo y reproducible.

En el panel de arriba a la izquierda tenemos esencialmente un editor de texto. Ahí es donde vas a escribir si querés guardar instrucciones para ejecutarlas en otro momento y donde vas a estar el 87 % de tu tiempo usando R.

A la derecha hay paneles más bien informativos y que tienen varias solapas que vamos a ir descubriendo a su tiempo. Para destacar, arriba a la derecha está el "environment", que es forma de ver qué es lo que está "pensando" R en este momento. Ahí vas a poder ver un listado de los datos que están abiertos y otros objetos que están cargados en la memoria de R. Ahora está vacío porque todavía no cargaste ni creaste ningún dato. Abajo a la derecha tienen un explorador de archivos rudimentario y también el panel de ayuda, que es donde vas a pasar el otro 13 % del tiempo usando R.

Entonces, para resumir:

1.2.2. Hablando con R

Ya viste cómo usar R como una calculadora.

```
2 + 2
```

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

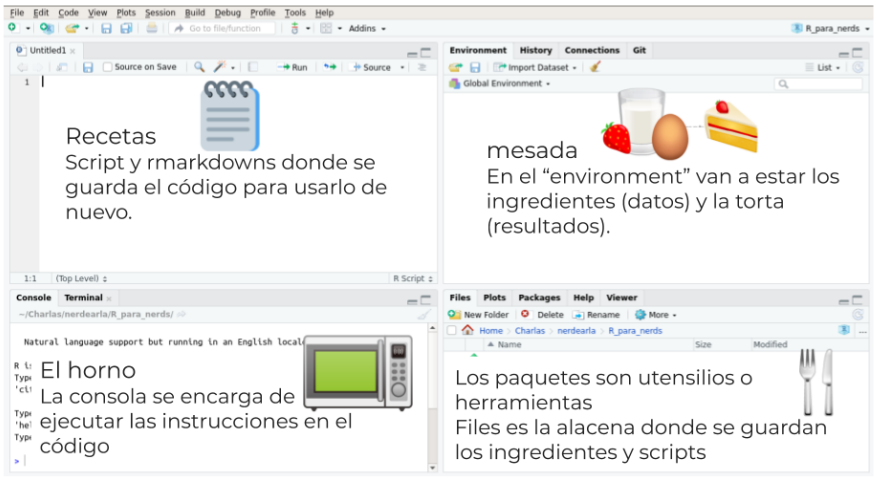


Figura 1.3: La cocina de RStudio

Si usaste fórmulas en Excel, esto es muy parecido a poner `=2+2` en una celda. R entiende un montón de operaciones aritméticas escritas como seguramente ya te imaginás:

- `+`: sumar
- `-`: restar
- `*`: multiplicar
- `/`: dividir
- `^` o `**`: exponenciar

Pero además conoce muchas otras operaciones. Para decirle a R que calcule el seno de 1 hay que escribir esto:

```
sin(1)
```

```
## [1] 0.841471
```

Esto es similar a poner `=SIN(1)` en Excel. La sintaxis básica para aplicar cualquier función es `nombre_funcion(argumentos)`.

Nota: En Excel el nombre de las funciones dependen del idioma en el que está instalado. Si lo usás en español, la función seno es `SEN()`. En R, las funciones siempre se escriben igual (que coincide con el inglés).

Desafío

Decile a R que compute las siguientes operaciones:

- 2 multiplicado por 2
- 3 al cuadrado
- dos tercios
- 5 por 8 más 1

Al hacer todas estas operaciones, lo único que hiciste fue decirle a R que haga esos cálculos. R te devuelve el resultado, pero no lo guarda en ningún lado. Para decirle que guarde el resultado de una operación hay que decirle con qué "nombre" querés guardarlo. El siguiente código hace eso:

```
x <- 2 + 2
```

La "flechita" `<-` es el operador de asignación, que le dice a R que tome el resultado de la derecha y lo guarde en una variable con el nombre que está a la izquierda. Vas a ver que no te devele el resultado. Para verlo, ejecutamos

```
x
```

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

Esto le dice a R que te "imprima" el contenido de la variable x.

Desafío

¿Qué te imaginás que va a pasar cuando ahora corra el siguiente código?

```
x + 2
```

Ponerle nombre a las variables es a veces la parte más difícil de escribir código. A R le viene bien cualquier nombre de variable siempre y cuando no empiece con un número o un "_". Pero a los seres humanos que lean el código y tengan que interpretarlos les va a resultas más fácil entender qué hace la variable `promedio_noches_estadia` que la variable `xxy1`.

El consejo es tratar en lo posible usar nombre descriptivos y consistentes. Por ejemplo, siempre usar minúsculas y separar palabras con "_".

Tip: Para hacerse la vida más fácil existen "guías de estilo" para programar que explicitan reglas específicas para escribir código. Por ejemplo esta o esta otra. Se trata de reglas únicamente para los ojos humanos, y que no afectan en absoluto la eficiencia o correctitud de la programación. En general, no existen guías buenas o malas, la idea es elegir una y ser consistente. De esta manera, vas a poder entender tu código con más facilidad.

1.2.3. Extendiendo R

R es un lenguaje creado por personas que practican la estadística y pensado para la estadística, por lo que ya viene con un montón de métodos estadísticos incorporados, como `mean()` o `median()`. Pero hay tantos métodos estadísticos como gente haciendo estadística así que es imposible que estén todos. La solución es que podés "agregarle" a R funciones que no vienen instaladas por defecto pero que escribieron otras personas en forma de "paquetes". ¡Este es el poder de **la comunidad de R**!

Para instalar paquetes de R, la forma mas fácil es con la función `install.packages()`. Esta función se conecta a internet y descarga paquetes publicados en un repositorio oficial Entonces, por ejemplo,

```
install.packages("readr")
```

descarga e instala un paquete que contiene funciones para leer datos.

Nota: Para instalar paquetes de esta forma es necesario tener conexión de internet.

Luego, usando el comando

```
library(readr)
```

le decís a R que cargue las funciones que vienen en el paquete readr para usarlas.

Desafío: Instalá el paquete readr con el comando `install.packages("readr")` en la consola.

Nota: Si cerrás y volveś a abrir R, vas a tener que usar `library(readr)` nuevamente para acceder a la funcionalidad del paquete readr. Sólo hace falta correr `install.packages("readr")` una vez por máquina.

1.2.4. Buscando ayuda

Entre la enorme cantidad de funciones que tiene R por defecto y las que se pueden agregar instalando paquetes externos, es imposible recordar todas las funciones y cómo usarlas. Por eso, una gran proporción del tiempo que uses R vas a pasarlo leyendo documentación de funciones, ya sea para aprender a usarlas o porque no te acordás algún detalle.

Para acceder a la ayuda de una función usamos el signo de pregunta:

```
?sin
```

Nota: Otra forma de acceder a la ayuda de una función es poniendo el cursor sobre ella y apretando F1

Esto va a abrir el documento de ayuda para la función `sin()` que, como verás, tiene la documentación de las funciones trigonométricas que trae R por defecto. Todas las ayudas de R vienen divididas en secciones:

Description Una descripción breve de la función o funciones que se documentan.

Usage Nombre de los argumentos de la función. La mayoría de las funciones trigonométricas tienen un solo argumento, que se llama `x`. La función `atan2()` tiene dos argumentos, llamados `x` e `y`.

Arguments Una descripción de cada argumento. En este caso `x` e `y` son vectores numéricos o complejos. Aunque todavía no sepas qué es un "vector", de esta descripción ya podés intuir que las funciones trigonométricas aceptan números complejos.

Details Una descripción detallada de las funciones. Por ejemplo, detalla qué es lo que devuelve la función `atan2()`, describe las unidades en las que tienen que estar `x` e `y`, etc..

Value Describe qué tipo de valor devuelve la función.

Examples (abajo de todo) Es la sección más importante y probablemente la que vas a buscar primero cuando te encuentres con una función nueva que no sabés cómo usar. Acá vas a encontrar código de R de que ejemplifica el uso típico de la función. Podes copiar y pegar el código en la consola y ver el resultado para entender como funciona.

(Otras secciones) Pueden haber otras secciones que detallen distintas particularidades de la función, o referencias a los métodos implementados.

Desafío

Abrí y leé la ayuda de la función `sd()`. Puede que haya cosas que aún no entiendas, pero tratá de captar la idea general. ¿Qué hace esa función? ¿Qué argumentos acepta?

Capítulo 2

Trabajar con proyectos en RStudio

Trabajar con proyectos de RStudio no solo hace tus análisis más ordenados y reproducibles, también hacen tu vida más simple.

Al comienzo posiblemente tengas un script y uno o dos archivos con datos, pero es posible que rápidamente te encuentres con una docena de archivos con nombres parecidos pero que pertenecen a análisis totalmente distintos. Antes de que la cosa comience a complicarse te proponemos trabajar con proyectos.

2.1. ¿Qué ventajas tiene?

- Te permite "cuidar" los datos que usas al ordenarnos en carpetas que diferencien entre la versión original o cruda y los datos limpios o los resultados finales.
- Te permite compartir tu trabajo fácilmente con otras personas. Solo tendrías que compartir la carpeta del proyecto sabiendo que incluye todo lo necesario para que cualquiera reproduzca tu análisis.
- Te permite publicar de manera ordenada tu código si vas a presentar o publicar tu trabajo.
- Te permite continuar con lo que estabas haciendo hace una semana o hace un mes como si el tiempo no hubiera pasado. De alguna manera es un regalo para tu yo futuro.

Primer desafío: Crea un nuevo proyecto en RStudio

1. Hacé click en el menú "Archivo" ("File") y luego en "Nuevo Proyecto" ("New Project").

2. Hacé click en "Nueva Carpeta" ("New Directory").
3. Hacé click en "Nuevo Proyecto" ("New Project").
4. Escribí el nombre de la carpeta que alojará a tu proyecto, por ejemplo "mi_proyecto"
5. Si aparece (y sabés usarlo), seleccioná "Crear un repositorio de git" ("Create a git repository").
6. Hacé click en "Crear Proyecto" ("Create Project").

Si todo salió bien, ahora deberías tener una nueva carpeta que se llama *mi_proyecto*. Pero si bien es una carpeta común y corriente, le llamamos proyecto porque además contiene un archivo con el mismo nombre *mi_proyecto.Rproj* (o solo *mi_proyecto* si en tu computadora no ves la extensión de los archivos).

2.2. Abrir un proyecto

La manera más simple de abrir un proyecto es abriendo la carpeta que lo contiene y haciendo doble click sobre el archivo *mi_proyecto.Rproj*. Al hacer esto se abrirá RStudio y la sesión de R en la misma carpeta y, por defecto, cualquier archivo que quieras abrir o guardar lo hará en esa misma ubicación. Esto ayuda a mantener tu trabajo ordenado y que luego sea simple retomar o compartir lo que hiciste.

RStudio permite tener varios proyectos abiertos, y esto es posible porque justamente cada proyecto tiene su propia carpeta. Si en algún momento trabajas con proyectos en paralelo vas a poder hacerlo sin que el código o los resultados de un análisis interfieran con otro.

Segundo desafío: Abrí tu nuevo proyecto desde el explorador de archivos

1. Cerrá RStudio
2. Desde el explorador de archivos, buscá la carpeta donde creaste tu proyecto.
3. Hacé doble click en el archivo que tiene el nombre de tu proyecto (y que termina con *.Rproj*) que encontrarás en esa carpeta.

2.3. ¿Cómo se organiza?

No existe una "mejor" forma de organizar un proyecto pero acá van algunos principios generales que nos hacen la vida más simple::

- **Tratar los datos como sólo de lectura** Es posible que la toma de los datos que querés analizar te haya costado mucho trabajo, o te haya costado conseguirlos. Trabajar con datos de forma interactiva (por ejemplo, en

Excel) tiene la ventaja de permitirte hacer algunos análisis rápidamente pero al mismo tiempo tiene la desventaja de que esos datos pueden ser modificados fácilmente. Esto significa que a veces no conozcas de la procedencia de los datos, o no recuerdes cómo los modificaste desde que los obtuviste. Por lo tanto, es una buena idea tratar los datos como "sólo de lectura" y nunca modificar los archivos originales.

- **Limpieza de datos** En muchos casos tus datos estarán "sucios", necesitarán un preprocesamiento importante para organizarlos en un formato que R (o cualquier otro lenguaje de programación) pueda analizados fácilmente. Esta tarea se denomina a veces "amasado" o "masticado de datos". Es una buena costumbre guardar el código que te permitió limpiar estos datos por si los volvieras a necesitar. También es recomendable guardar esa versión de los datos limpios, de "sólo lectura", para que puedas usarlos en tu análisis sin necesidad de repetir cada vez todo el proceso de limpieza de los datos.
- **Tratar las salidas o resultados generados como descartables** Cualquier resultado (gráficos, tablas, valores) debe poder repetirse o rehacerse a partir del código guardado. Si bien las pruebas rápidas para *ver si el código funciona* se pueden hacer en la consola, es importante guardar el código que genera los resultados y asegurarnos de que sean reproducibles. Aún mejor, si organizas esos resultados en distintas sub-carpetas, luego tendrás todo aún más ordenado.

2.4. Ordenando aún más

Si tenés alguna experiencia programando con R es posible que tengas estás lineas al comienzo de alguno de tus scripts o si nunca las usaste, seguro viste que alguien más lo hacia:

```
setwd("/Users/pao/una_carpeta/al/proyecto_importante")
rm(list = ls())
```

La primera línea *setea* o le avisa a R cual será la carpeta donde va a trabajar. *Con el uso de proyectos esto está prácticamente solucionado* porque al abrir el proyecto ya sea desde el explorador de archivos haciendo doble click en el archivo con extensión *Rproj* o desde RStudio, R sabrá que ese directorio será el de trabajo.

Pero también te dijimos que era una buena práctica organizar las diferentes partes del proyecto en subcarpetas, como por ejemplo colocar los datos en una subcarpeta llamada "datos", los informes en otra y tal vez las figuras en una subcarpeta distinta dentro del proyecto. ¿Cómo hacemos para que R lea un archivo que no está en la carpeta de trabajo? Podríamos escribir el camino hacia ese archivo, por ejemplo `"datos/mi_archivo_de_datos.csv"` pero si queremos

compartir el código a otra persona que tal vez tiene un sistema operativo distinto y usa la barra invertida `\` va a estar en problemas al intentar correr esa línea.

Para solucionar estos problemas existe el paquete `{here}`, que funciona independientemente del sistema operativo. Su función principal `here()` recibe como argumentos el camino hacia el archivo que se quiere leer, siempre entre comillas y separados por comas, así:

```
mis_datos <- read_csv(here("datos", "mi_archivo_de_datos.csv"))
```

Internamente este paquete puede identificar cual es el directorio de trabajo (por ejemplo detectando que hay un archivo *.Rproj*) y busca a partir de ahí la subcarpeta "datos" y adentro de ella el archivo "mi_archivo_de_datos.csv".

La segunda línea del código inicial se usa para borrar los elementos que creamos en el análisis normalmente cuando cambiamos de tema o empezamos a trabajar con algo distinto. Esto está bien porque no queremos arrastrar análisis que hicimos en un proyecto a otro, necesitamos que sean autocontenidos y *reproducibles*. El problema es que este comando **no** borra los paquetes activados o las opciones usuario que hayamos seteado.

2.4.1. Borrón y cuenta nueva... todos los días!

¿Cómo nos aseguramos de que el análisis sea realmente reproducible? Esta es una pregunta bastante amplia y hay muchas herramientas para resolver este problema. Por ahora nos vamos a concentrar en que al menos en tu computadora puedas repetir los cálculos o el análisis desde cero. Y además de organizar proyectos y no modificar los datos originales, ¿cómo podés asegurarte de que guardaste todo el código que estuviste escribiendo y usaste? La manera más directa es reiniciar la sesión de R y correr el código de nuevo, si da error o no devuelve lo que esperabas significa que te faltó guardar algún paso.

Tip: Podés reiniciar la sesión de R con el atajo `Ctrl+Shif+F10`

Esto puede pasar si por ejemplo leés una base de datos en memoria pero no guardás el código que lo hace. Mientras estemos trabajando, R tendrá esa base de datos en memoria y podremos hacer cálculos y gráficos. Por defecto además RStudio va a recordar las variables que estés usando mañana o pasado en un archivo oculto (.RData) a menos que le indiques lo contrario. Y si bien suena práctico volver a R al otro día y tener el análisis tal cual lo dejamos, esto puede significar que nunca nos demos cuenta que nos faltó guardar una línea de código clave en nuestro análisis.

Tercer desafío: Configurá RStudio

1. Hacé click en el menú "Herramientas ("Tools") y luego"Opciones globales" ("Global Options").

2. Destildá la opción “Recuperar .RData al inicio de la sesión” (“Restore .RData into workspace at startup”).
3. Hacé click en “Aplicar” (“Apply”).

Capítulo 3

Introducción a RMarkdown

Es posible que en tu trabajo tengas que presentar informes o resultados de tu análisis de datos. Tal vez te hayas encontrando guardando una y otra vez gráficos y tablas o copiando resultados de un archivo al otro hasta que el informe quedó como querías. Los archivos y el paquete **RMarkdown** vienen al rescate.

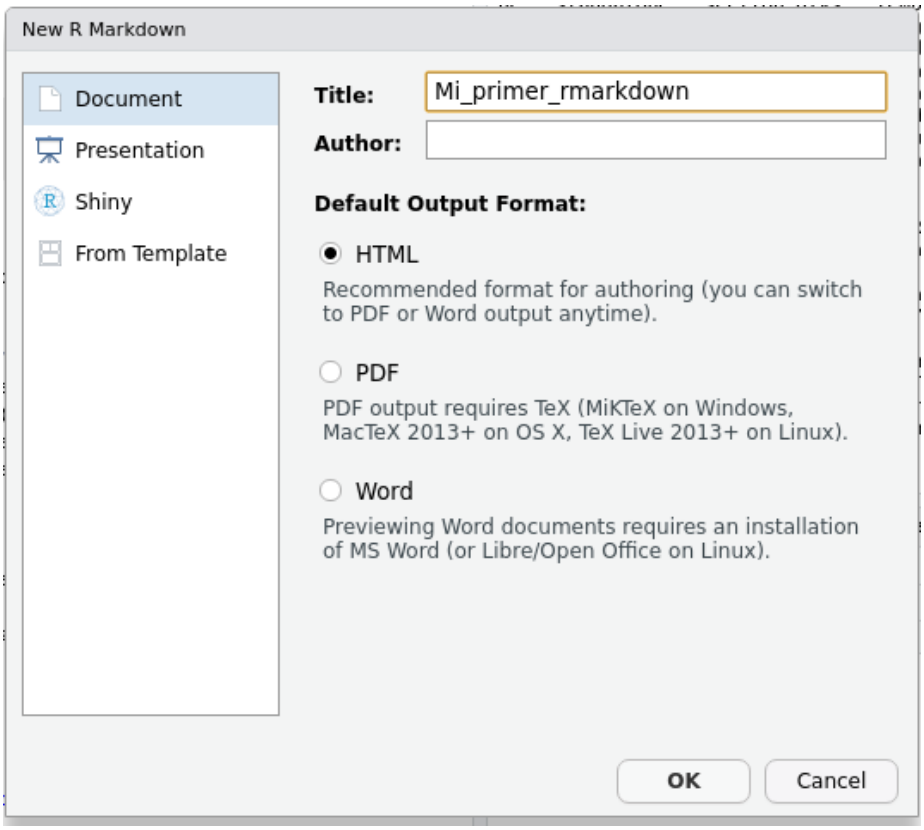
Un archivo de R Markdown (generalmente con la extensión `.Rmd`), a diferencia de un script `.R`, es un archivo de texto plano que combina código de R que genera resultados (gráficos, tablas, etc…) y el texto que lo describe. Al poder intercalar cálculos y gráficos con su análisis o explicación, se unifica el flujo de trabajo y deja de ser necesario guardar figuras o tablas para luego insertarlas en un documento de texto. Esto es muy importante si buscamos que nuestro trabajo sea reproducible, pero además ahorra tiempo.

3.1. Creando archivos .Rmd

En RStudio podés crear un nuevo archivo de R Markdown con el menú desplegable:

File → New File → R Markdown

Y se abrirá un menú donde podés agregar el título de tu informe y tu nombre. Por ahora vamos a usar el formato HTML como salida, pero más adelante vas a ver que hay muchos otros formatos de salida posibles.



Al aceptar, se abrirá un nuevo archivo con una plantilla de ejemplo (en inglés).

Primer desafío: Creá un nuevo archivo R Markdown

Revisá la plantilla que trae el documento. ¿Podés identificar los bloques de código?

Para generar el archivo de salida, el paquete **knitr** (que viene de *tejer* en inglés) ejecutará el código en una sesión independiente de R e interpretará el texto, su formato y cualquier otra cosa que agreguemos (por ejemplo imágenes o links externos). Esto significa que nuestro archivo debe tener **todo** lo necesario para generar el análisis y si nos olvidamos de algo va a dar error.

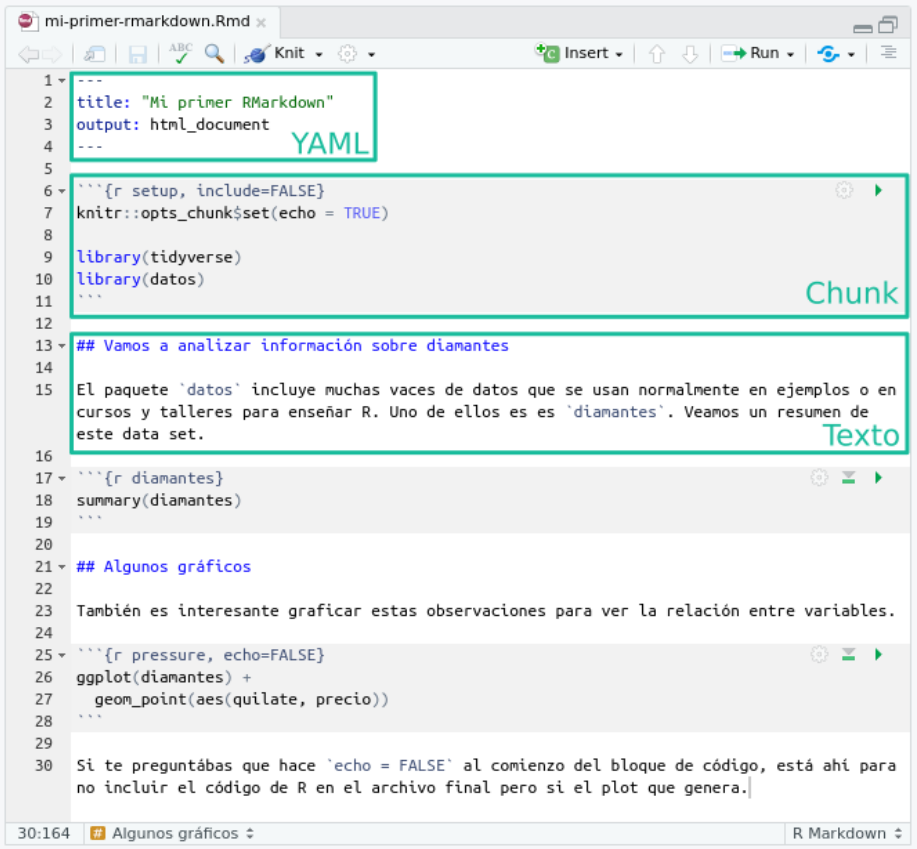
Por esta razón es recomendable *knitear* el archivo seguido, para encontrarnos con los errores a tiempo y de paso asegurarnos que el análisis es reproducible.

Segundo desafío: kniteá tu R Markdown

Aprovechando la plantilla de RStudio, obtené el archivo de salida en formato HTML haciendo click en el botón **knit** (el que tiene un ovillo de lana y un par de agujas!).

3.2. Estructura de un .Rmd

Cualquier archivo de este tipo tiene 3 partes principales:



(Podés encontrar este archivo de ejemplo acá.)

3.2.1. Encabezado

El encabezado es una serie de instrucciones organizadas entre tres guiones (---) que determinan las propiedades globales del documento, como el título, el formato de salida, información de autoría, etc… También ahí se pueden cambiar opciones asociadas al formato de salida, como el estilo de la tabla de contenidos o índice.

Éstas propiedades se definen en un formato llamado YAML, el cual permite definir listas jerarquizadas de una forma humanamente legible. Por ejemplo:

```
---
title: "Mi primer RMarkdown"
output:
  html_document:
    code_download: true
    toc: true
    toc_float: false
---
```

define dos variables principales, “title” y “output”. “Output” a su vez contiene un elemento “htm_document”, el cual contiene tres elementos: “code_download”, “toc” y “toc_float”.

Es muy importante mantener el escalonado, o *identación* de los elementos, ya que ésta define la jerarquía de cada elemento. Muchos de los errores a la hora de knitear ocurren porque el archivo tiene problemas en la identación del encabezado.

3.2.2. Bloques de código

El código de R que va a leer datos, analizarlos y generar figuras, tablas o números se organiza en bloques (o `chunks`) delimitados por tres acentos graves (```) y se diferencia del resto de archivo con un fondo gris. Todo lo que incluyas entre estos delimitadores será interpretado por R como código e intentará ejecutarlo al *knitear* el archivo. Cualquier resultado del código (gráficos, tablas, texto, etc…) será insertado en el documento final en el mismo orden que están en el archivo R Markdown.

Para insertar un nuevo chunk podés:

- Usar el botón **Insert**
- El atajo de teclado Ctrl+Alt+I
- Escribir a mano!

El código en cada bloque se ejecuta como si fuera ejecutado en la terminal y todo resultado se muestra en el documento (ya vamos a ver formas de controlar esto). Por ejemplo, este bloque de código

````
```{r sumar}
1 + 1
```
````

va a insertar esto en el documento de salida:

```
[1] 2
```

Es muy importante no romper los límites de los bloques. Un problema común es accidentalmente eliminar un acento grave al final de un bloque de código y que luego el documento no knitee correctamente. Si al knitear te sale un error como "attempt to use zero-length variable name", revisá bien que todos tus bloques de código estén correctamente definidos.

Los bloques pueden tener nombre, lo cual es útil para identificar donde ocurren los errores al momento de *knitear* pero también para tener una pista de lo que hace el código que incluye.

Si bien el código se corre cuando uno knitea, cuando estés escribiendo un informe es muy cómodo ir corriendo bloques individuales interactivamente como si fuera en la consola.

Para correr una línea de código, tendrás que pararte sobre esa línea y apretar:

Ctrl+Enter

Pero también podés correr el código de todo el chunk con:

Ctrl+Shift+Enter

Los resultados van a aparecer inmediatamente debajo del bloque.

Cuarto desafío: Sumá un chunk a tu archivo

Usando el archivo con el que venís trabajando insertá un nuevo chunk y:

1. Cargá el paquete readr.
2. Creá una variable que se llame `variable_prueba` y asignale un valor.
3. Mostrá ese valor.
4. Volvé a *knitear* el archivo para ver el resultado

Finalmente, es posible que te encuentres mencionando resultados en el texto, por ejemplo algo así como "el porcentaje de ocupación para el mes de enero fue del 95 %". Y también es posible que ese valor cambie si utilizas una base de datos distinta o si luego generas un informe pero para un mes siguiente. Las chances de de que te olvides de actualizar ese "95" son super altas, por eso R Markdown también tiene la posibilidad de incorporar código en línea con el texto.

Si tenés una una variable `ocupacion` que vale "95 %":

```
ocupacion <- "95%"
```

Para mencionarla en el texto entonces escribirías:

> El porcentaje de ocupación para el mes de enero fue del `` `r ocupacion` ``.

y el resultado en el documento kniteado sería

El porcentaje de ocupación para el mes de enero fue del 95 %.

prueba: `95 %`

3.2.3. El texto propio del documento.

Este es el texto dirigido a las personas que van a leer el reporte. Incluirá una introducción, descripción de los datos y de los resultados; es lo que escribirías en el archivo de Word.

A diferencia de Word, el formato del texto se define usando markdown, que es un lenguaje simple que permite indicar si un texto va en negrita, cursiva, es un título, etc…usando símbolos especiales dentro del texto.

3.3. Markdown

Markdown permite escribir en texto plano pero definiendo el formato usando símbolos. Por ejemplo podés resaltar con **negrita** usando dos asteriscos así: `**negrita**` o *italizada* con un asterisco de cada lado: `*itálicas*`.

También podés hacer una lista de elementos utilizando asteriscos:

```
* la negrita se consigue con dos asteriscos
* la italizada con un asterisco
* y para resaltar código se usa el acento grave `
```

o guiones medios:

```
- la negrita se consigue con dos asteriscos
- la italizada con un asterisco
- y para resaltar código se usa el acento grave `
```

Ambas listas se van a ver de esta manera:

- la negrita se consigue con dos asteriscos
- la italizada con un asterisco
- y para resaltar código se usa el acento grave ‘

Si en realidad querés una lista numerada, simplemente comenzá el renglón un número y un punto. Podrías usar siempre el mismo número, markdown se encarga del resto:

```
1. la negrita se consigue con dos asteriscos
1. la italizada con un asterisco
1. y para resaltar código se usa el acento grave `
```

Ahora la lista numerada se ve así:

1. la negrita se consigue con dos asteriscos
2. la italizada con un asterisco
3. y para resaltar código se usa el acento grave ‘

Podés agregar títulos con distinta jerarquía agregando `#` al comienzo. Esto además define secciones dentro del documento:

```
# Título
## El primer subtítulo
### Otro subtítulo de menor jerarquía
#### Otro más, y podría seguir!
```

Podés escribir estos símbolos a mano o usando el Editor Visual de RStudio (sólo disponible para la versión 1.4 en adelante) haciendo click en el ícono de compás que está a la derecha del documento () . El Editor Visual permite dar formato al texto usando markdown sin saber usar markdown.

Tercer desafío: Agregale texto a tu archivo

Borrá el contenido del archivo `.Rmd` que creaste (pero no el encabezado!) y probá escribir algo y darle formato. Luego volvé a apretar el botón **knit** para ver el resultado.

Markdown permite muchas otras cosas, por ejemplo:

Podés agregar un link a una página externa: `[texto que se muestra con el link](http://google.com)`. Resultado: texto que se muestra con el link

Podés incluir una imagen: `![descripción de la figura](https://placekitten.com/200/100)`: Resultado:

Figura 3.1: descripción de la figura

Y también podés agregar ecuaciones (usando LaTeX) en la misma línea (esto:`$E = mc^2$` se ve así: $E = mc^2$) o en una línea propia. Esto:

```
$$
y = \mu + \sum_{i=1}^p \beta_i x_i + \epsilon
$$
```

se ve así:

$$y = \mu + \sum_{i=1}^{p} \beta_i x_i + \epsilon$$

Podés revisar la guía rápida de Markdown desde RStudio (en inglés):

Help → Markdown Quick Reference

Capítulo 4

Lectura de datos ordenados

4.1. Descargando datos

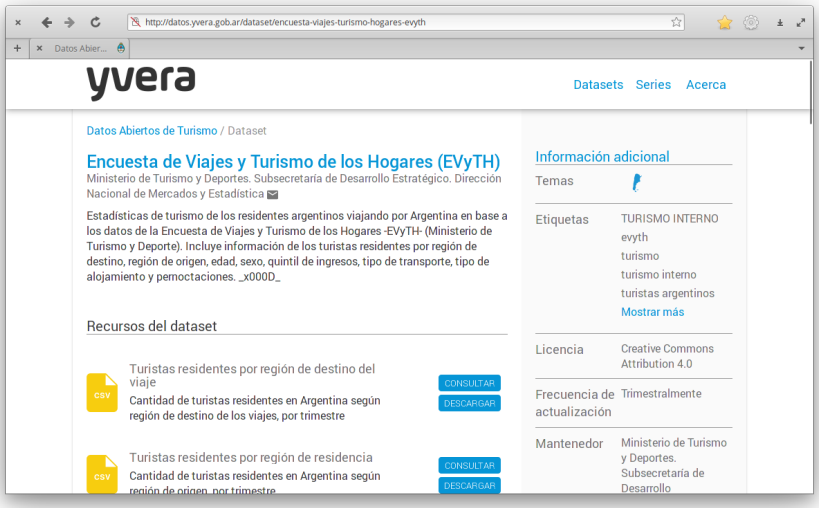
Antes de poder leer cualquier dato en R, primero hay que encontrarlo y descargarlo. El Ministerio de Turismo mantiene un portal de datos abierto llamado Yvera donde podés buscar y descargar datos relacionados con el turismo en Argentina.

En esta sección vamos a descargar una serie de tiempo a partir de la Encuesta de Viajes y Turismo de los Hogares (EVyTH).

Primero, ingresá a http://datos.yvera.gob.ar/, donde te vas a encontrar con la páigna principal de Yvera.

La base de datos que vamos a descargar está en el área de Turismo Interno así que hacé click en ese ícono para navegar a la sección de datasets correspondiente.

Al momento que escribimos esta guía el primer set de datos que aparece es la Encuesta de Viajes y Turismo de los Hogares (EVyTH). Hacé click ahí para ir a la página de este set de datos.



Este set de datos tiene distintos recursos. Varios son datos, como "Turistas residentes por región de destino del viaje" o "Turistas residentes por edad", pero también hay un recurso llamado "Ficha Técnica: EVyTH". Este es un PDF con la descripción de los datos así como consideraciones metodológicas relevantes. Es importante que si vas a usar datos siempre mires la ficha técnica para hacerte una idea de las limitaciones metodológicas que pueden tener estos datos.

Por ahora vamos a descargar la serie de Turistas residentes por edad.

En esta pantalla te vas a encontrar con una descripción del set de datos y sus variables. Para descargar los datos hay que hacer click en el botón que dice "DESCARGAR". Guardalo en una carpeta dentro del proyecto (recomedamos organizar los datos en una carpeta llamada "datos") y ya está listo para leer.

Pero si tuvieras que realizar un informe mensual sobre estos datos tendrías que hacer toda esta descarga manual cada vez que se actualiza el informe. La gracia de usar código es automatizar todo lo más posible, así que en vez de descargar manualmente el archivo, se puede descargar automáticamente desde el código de R.

Para eso, primero necesitás la dirección (URL) del set de datos. Eso se consigue yendo a la página del set de datos y en vez de hacer click en DESCAGAR, haciendo

Click derecho → Copiar dirección del enlace

La URL de la esta serie es `http://datos.yvera.gob.ar/dataset/945e10f1-eee7-48a2-b0ef-8aff11df8814/resou`
Guaramos eso en una variable en R

```
turistas_edad_url <- "http://datos.yvera.gob.ar/dataset/945e10f1-eee7-48a2-b0ef-8aff11df8814/reso
```

Y también definimos la ruta donde descargar el archivo

```
turistas_edad_archivo <- "datos/turistas_edad.csv"
```

Y finalmente usamos la función `download.file()` para descargar el archivo.

```
download.file(url = turistas_edad_url, destfile = turistas_edad_archivo)
```

Y esto va a descargar la última versión de los datos.

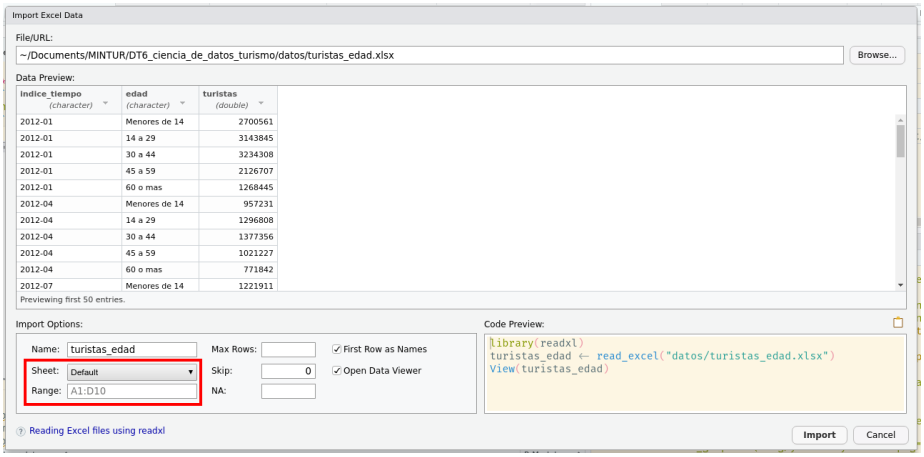
4.2. Leer datos csv

Existen muchas funciones distintas para leer datos dependiendo del formato en el que están guardados. Para datos tabulares, la forma más útil es el formato csv, que es un archivo de texto plano con datos separados por coma.

Para importar datos hace falta escribir el código correspondiente pero también podés aprovechar el entorno gráfico de RStudio:

File → Import Dataset → From Text (readr)...

Esto te va abrir una ventana donde podrás elegir el archivo a importar (en este caso el archivo `turistas_edad.csv` que está dentro de la capeta `datos` del proyecto) y otros detalles.



En la pantalla principal vas a poder previsualizar los datos y ver que pinta tienen. Abajo a la izquierda tenés varias opciones: el nombre que vas a usar para la variable (en este caso llamaremos `turistas_edad`), si la primera fila contiene los nombres de las columnas (`First Row as Names`), qué delimitador tienen los datos (en este caso `comma`, pero podría ser punto y coma u otro), etc...

Y abajo a la derecha es el código que vas a necesitar para efectivamente importar los datos. Podrías apretar el botón "Import" para leer los datos pero si bien es posible, al mismo tiempo esas líneas de código no se guardan en ningún lado y entonces nuestro trabajo luego no se puede reproducir. Por eso, te proponemos

que copies ese código, cierres esa ventana con el botón "Cancel", y pegues el código en el archivo donde estés trabajando. Cuando lo ejecutes, se va a generar la variable `turistas_edad` con los datos.

```
library(readr)
turistas_edad <- read_csv("datos/turistas_edad.csv")
```

```
## Rows: 180 Columns: 3
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): indice_tiempo, edad
## dbl (1): turistas
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Nota: Notá que en este caso el código para leer los datos consta de dos líneas. La primera carga el paquete **readr** y el segundo usa la función `read_csv()` (del paquete readr) para leer el archivo .csv. No es necesario cargar el paquete cada vez que vas a leer un archivo, pero asegurate de incluir esta línea en el primer bloque de código de tu archivo.

Nota: La interfaz de usuario de RStudio sirve para autogenerar el código que lee el archivo. Una vez que lo tenés, no necesitás abrirla de nuevo.

Todo ese texto naranja/rojo es intimidante pero no te preocupes, es sólo un mensaje que nos informa que los datos se leyeron y qué tipo de dato tiene cada columna. Podemos explorar la estructura de la variable `turistas_edad` usando la función `str()` (de *structure* en inglés).

```
str(turistas_edad)
```

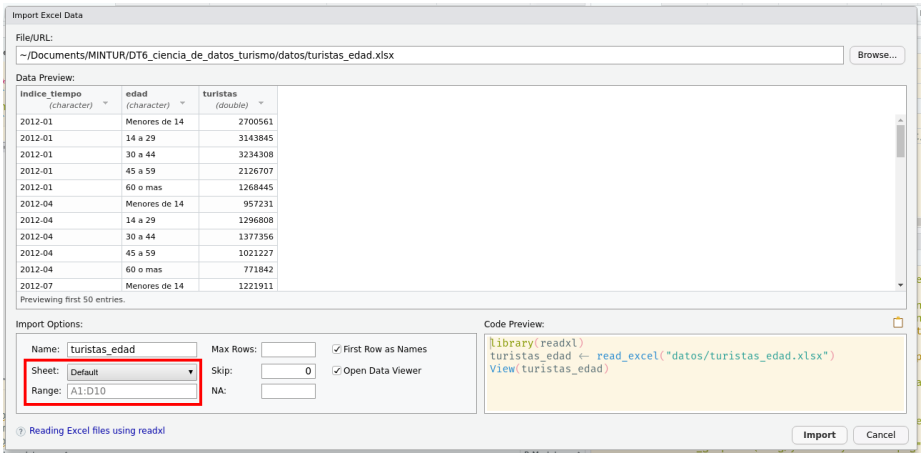
```
## spec_tbl_df [180 x 3] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ indice_tiempo: chr [1:180] "2012-01" "2012-01" "2012-01" "2012-01" ...
##  $ edad         : chr [1:180] "Menores de 14" "14 a 29" "30 a 44" "45 a 59" ...
##  $ turistas     : num [1:180] 2700561 3143845 3234308 2126707 1268445 ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   indice_tiempo = col_character(),
##   ..   edad = col_character(),
##   ..   turistas = col_double()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

Esto nos dice un montón. La primera línea dice que es una `tibble`, que es un caso especial de la estructura de datos tabular básica de R llamada `data.frame`. Tiene 180 filas (las **observaciones**) y 3 columnas (o **variables** que describen las observaciones). Las siguientes líneas nos dicen los nombres de las columnas (indice_tiempo, edad, y turistas), su tipo de dato (`chr` o `num`), la longitud ([1:180]) y sus primeros elementos.

4.3. Leer datos de excel

Si tenés la vista avispada, habrás notado que en el menú de "Import Dataset" hay una opción para leer datos de Excel. En efecto, RStudio provee la misma ayuda para leer este tipo de datos:

File → Import Dataset → From Excel...



Notá que entre las opciones de abajo a la izquierda aparecen dos variables importantes. Podés seleccionar de qué hoja leer los datos y qué rango usar. Esto seguro que te va a ser muy útil para esos archivos de Excel con múltiples tablas en un archivo, o incluso múltiples tablas en cada hoja!

En este caso turistas_edad.xlsx es un Excel buena onda, y el código para leer los datos es muy simple:

```
library(readxl)
turistas_edad <- read_excel("datos/turistas_edad.xlsx")
```

Con la función `str()` podés confirmar que los datos leídos son los mismos que para el csv.

```r
str(turistas_edad)
```

```
## tibble [180 x 3] (S3: tbl_df/tbl/data.frame)
##  $ indice_tiempo: chr [1:180] "2012-01" "2012-01" "2012-01" "2012-01" ...
##  $ edad         : chr [1:180] "Menores de 14" "14 a 29" "30 a 44" "45 a 59" ...
##  $ turistas     : num [1:180] 2700561 3143845 3234308 2126707 1268445 ...
```

Desafío: Lee un archivo

1. Lee el archivo `turistas_edad.xlsx`, pero solo las primeras 30 lineas
2. ¿Qué cambió en código que devuelve RStudio?
3. Revisa la documentación de la función `read_excel()` para identificar otros argumentos que puedan resultarte útiles.

Capítulo 5

Manipulación de datos ordenados

El paquete **dplyr** provee una enorme cantidad de funciones útiles para manipular y analizar datos de manera intuitiva y expresiva.

El espíritu detrás de dplyr es que la enorme mayoría de los análisis, por más complicados que sean, involucran combinar y encadenar una serie relativamente acotada de acciones (o verbos). En este curso vamos a centrarnos las cinco más comunes:

- `select()`: selecciona columnas de una tabla.
- `filter()`: selecciona (o filtra) filas de una tabla a partir de una o más condiciones lógicas.
- `group_by()`: agrupa una tabla en base al valor de una o más columnas.
- `mutate()`: agrega nuevas columnas a una tabla.
- `summarise()`: calcula estadísticas para cada grupo de una tabla.

dplyr y tablas dinámicas:

A grosso modo, las operaciones de dplyr permiten hacer en R lo que se hace en tablas dinámicas (pivot tables) en Excel. `filter()` vendría a ser como la sección de "Filtros", `group_by()`, la sección de "Filas", `select()`, la sección de "Columnas" y `summarise()`, la parte de "Valores".

Primer desafío:

Te dieron una tabla con datos de cantidad mensual de visitantes residentes y visitantes no residentes para cada provincia. Las columnas son: `año`, `mes`, `provincia`, `visitantes_residentes` y `visitantes_no_residentes`. En base a esos datos, te piden que calcules la proporción promedio de visitantes que son residentes para cada provincia en enero.

¿En qué orden ejecutarías estos pasos para obtener el resultado deseado?

- usar `summarise()` para calcular la estadística `mean(proporcion_residentes)` para cada `provincia`
- usar `group_by()` para agrupar por la columna `provincia`
- usar `mutate()` para agregar una columna llamada `proporcion_residentes` que sea `visitantes_residentes/(visitantes_residentes + visitantes_no_residentes)`.
- usar `filter()` para seleccionar solo las filas donde la columna `mes` es igual a 1.

Para usar dplyr primero hay que instalarlo (esto hay que hacerlo una sola vez por computadora) con el comando:

```
install.packages("dplyr")
```

y luego cargarlo en memoria con

```
library(dplyr)
```

Volvé a cargar los datos de turistas por edad (para un recordatorio, podés ir a Lectura de datos ordenados):

```
library(readr)
turistas_edad <- read_csv("datos/turistas_edad.csv")
```

5.1. Seleccionando columnas con `select()`

Para quedarse únicamente con las columnas de índice de tiempo y turistas, usá `select()`

```
select(turistas_edad, indice_tiempo, turistas)
```

```
## # A tibble: 180 x 2
##    indice_tiempo turistas
##    <chr>            <dbl>
##  1 2012-01        2700561
##  2 2012-01        3143845
##  3 2012-01        3234308
##  4 2012-01        2126707
##  5 2012-01        1268445
##  6 2012-04         957231
##  7 2012-04        1296808
```

```
##  8 2012-04        1377356
##  9 2012-04        1021227
## 10 2012-04         771842
## # ... with 170 more rows
```

¿Dónde quedó este resultado? Si te fijás en la tabla `turistas_edad`, su formato no cambió, sigue teniendo todas las columnas originales a pesar de nuestro select:

```
turistas_edad
```

```
## # A tibble: 180 x 3
##    indice_tiempo edad          turistas
##    <chr>         <chr>            <dbl>
##  1 2012-01       Menores de 14  2700561
##  2 2012-01       14 a 29        3143845
##  3 2012-01       30 a 44        3234308
##  4 2012-01       45 a 59        2126707
##  5 2012-01       60 o mas       1268445
##  6 2012-04       Menores de 14   957231
##  7 2012-04       14 a 29        1296808
##  8 2012-04       30 a 44        1377356
##  9 2012-04       45 a 59        1021227
## 10 2012-04       60 o mas        771842
## # ... with 170 more rows
```

`select()` y el resto de las funciones de dplyr siempre generan una nueva tabla y nunca modifican la tabla original. Para guardar la tabla con las dos columnas `indice_tiempo` y `turistas` tenés que asignar el resultado a una nueva variable.

```
turistas_edad2 <- select(turistas_edad, indice_tiempo, turistas)
turistas_edad2
```

```
## # A tibble: 180 x 2
##    indice_tiempo turistas
##    <chr>            <dbl>
##  1 2012-01        2700561
##  2 2012-01        3143845
##  3 2012-01        3234308
##  4 2012-01        2126707
##  5 2012-01        1268445
##  6 2012-04         957231
##  7 2012-04        1296808
##  8 2012-04        1377356
##  9 2012-04        1021227
## 10 2012-04         771842
## # ... with 170 more rows
```

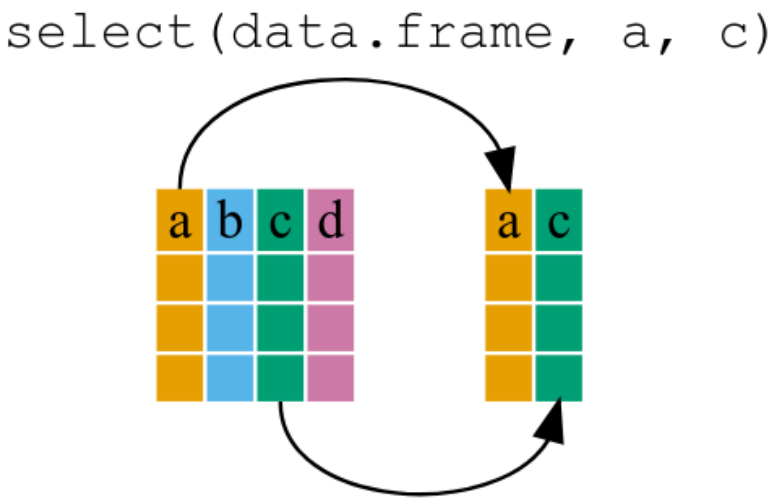


Figura 5.1: Cómo funciona `select()`

5.2. Filtrando filas con `filter()`

Ahora podés usar `filter()` para quedarte con sólo unas filas. Por ejemplo, para quedarse con los turistas menores de 14 años

```
filter(turistas_edad, edad == "Menores de 14")
```

```
## # A tibble: 36 x 3
##    indice_tiempo edad          turistas
##    <chr>         <chr>            <dbl>
##  1 2012-01       Menores de 14  2700561
##  2 2012-04       Menores de 14   957231
##  3 2012-07       Menores de 14  1221911
##  4 2012-10       Menores de 14  1172970
##  5 2013-01       Menores de 14  2604040
##  6 2013-04       Menores de 14   942360
##  7 2013-07       Menores de 14  1341895
##  8 2013-10       Menores de 14  1382380
##  9 2014-01       Menores de 14  2706215
## 10 2014-04       Menores de 14  1006238
## # ... with 26 more rows
```

La mayoría de los análisis consisten en varios pasos que van generando tablas intermedias (en el primer desafío usaste 4 pasos para calcular la proporción media de visitantes residentes) La única tabla que te interesa es la última, por lo que ir asignando variables nuevas en cada paso intermedio es tedioso y poco práctico. Para eso se usa el operador 'pipe' (`%>%`).

El operador 'pipe' (`%>%`) agarra el resultado de una función y se lo pasa a la siguiente. Esto permite escribir el código como una cadena de funciones que van operando sobre el resultado de la anterior.

Las dos operaciones anteriores (seleccionar tres columnas y luego filtrar las filas correspondientes a Argentina) se pueden escribir uno después del otro y sin asignar los resultados intermedios a nuevas variables de esta forma:

```
turistas_edad %>%
  filter(edad == "Menores de 14") %>%
  select(indice_tiempo, turistas)
```

```
## # A tibble: 36 x 2
##    indice_tiempo turistas
##    <chr>            <dbl>
##  1 2012-01        2700561
##  2 2012-04         957231
```

```
##  3 2012-07        1221911
##  4 2012-10        1172970
##  5 2013-01        2604040
##  6 2013-04         942360
##  7 2013-07        1341895
##  8 2013-10        1382380
##  9 2014-01        2706215
## 10 2014-04        1006238
## # ... with 26 more rows
```

La forma de "leer" esto es "Tomá la variable `turistas_edad`, **después** aplicale `filter(edad == "Menores de 14")`, **después** aplicale `select(indice_tiempo, turistas)`".

Cómo vimos, el primer argumento de todas las funciones de dplyr es el data frame sobre el cual van a operar, pero notá que en las líneas con `select()` y `filter()` no escribís la tabla explícitamente.
Esto es porque la pipe implícitamente pasa el resultado de las líneas anteriores como el primer argumento de la función siguiente.

Toma el data frame `turistas_edad` y se lo pasa al primer argumento de `select()`. Luego el data frame resultante de esa operación pasa como el primer argumento de la función `filter()` gracias al `%>%`.

Tip:

En RStudio podés escribir `%>%` usando el atajo de teclado Ctr + Shift + M. ¡Probalo!

Desafío:

Completá esta cadena para producir una tabla que contenga los datos de turistas de 60 o más años

```
turistas_edad %>%
  filter(edad == ___) %>%
  select(_____, ____)
```

5.3. Agrupando y reduciendo con `group_by() %>% summarise()`

Si querés calcular la cantidad promedio de turistas por cada rango de edad, tenés que usar el combo `group_by() %>% summarise()`. Es decir, primero agrupar la tabla según la columna `edad` y luego calcular el promedio de `turistas` para cada grupo.

Para agrupar la tabla países según el continente usamos el siguiente código:

```
turistas_edad %>%
  group_by(edad)
```

```
## # A tibble: 180 x 3
## # Groups:   edad [5]
##    indice_tiempo edad          turistas
##    <chr>         <chr>            <dbl>
##  1 2012-01       Menores de 14  2700561
##  2 2012-01       14 a 29        3143845
##  3 2012-01       30 a 44        3234308
##  4 2012-01       45 a 59        2126707
##  5 2012-01       60 o mas       1268445
##  6 2012-04       Menores de 14   957231
##  7 2012-04       14 a 29        1296808
##  8 2012-04       30 a 44        1377356
##  9 2012-04       45 a 59        1021227
## 10 2012-04       60 o mas        771842
## # ... with 170 more rows
```

A primera vista parecería que la función no hizo nada, pero fijate que el resultado ahora dice que tiene grupos ("Groups:"), y nos dice qué columna es la que agrupa los datos ("edad") y cuántos grupos hay ("5"). Las operaciones subsiguientes que le hagamos a esta tabla van a hacerse *para cada grupo*.

Para ver esto en acción, usá `summarise()` para computar el promedio de turistas

```
turistas_edad %>%
  group_by(edad) %>%
  summarise(turistas_promedio = mean(turistas))
```

```
## # A tibble: 5 x 2
##   edad          turistas_promedio
##   <chr>                     <dbl>
## 1 14 a 29                1440779.
## 2 30 a 44                1530064.
## 3 45 a 59                1242697.
## 4 60 o mas               1033486.
## 5 Menores de 14          1368313.
```

¡Tadá! `summarise()` devuelve una tabla con una columna para la edad y otra nueva, llamada "turistas_promedio" que contiene el promedio de `turistas` para cada grupo. Esta operación es equivalente a esta tabla dinámica en Excel:

`group_by()` permite agrupar en base a múltiples columnas y `summarise()` permite generar múltiples columnas de resumen. El siguiente código calcula la cantidad promedio de turistas y su desvío estándar para cada continente y cada año.

```
turistas_edad %>%
  group_by(edad, mes = substr(indice_tiempo, 6, 7)) %>%
  summarise(turistas_promedio = mean(turistas),
            turistas_desvio = sd(turistas))
```

```
## `summarise()` has grouped output by 'edad'. You can override using the `.groups` argument.
```

```
## # A tibble: 20 x 4
## # Groups:   edad [5]
##    edad          mes   turistas_promedio turistas_desvio
##    <chr>         <chr>             <dbl>           <dbl>
##  1 14 a 29       01             2432791.         439326.
##  2 14 a 29       04              889256.         421702.
##  3 14 a 29       07             1239470.         508496.
##  4 14 a 29       10             1201600.         310893.
##  5 30 a 44       01             2571253.         411656.
##  6 30 a 44       04              954923.         419869.
##  7 30 a 44       07             1289021.         454895.
##  8 30 a 44       10             1305058.         301415.
##  9 45 a 59       01             1955239.         119285.
## 10 45 a 59       04              826473.         319558.
## 11 45 a 59       07             1088687.         421866.
## 12 45 a 59       10             1100391.         245137.
## 13 60 o mas      01             1410029.         196945.
## 14 60 o mas      04              827572.         323722.
## 15 60 o mas      07              931603.         364683.
## 16 60 o mas      10              964739.         278363.
## 17 Menores de 14 01             2452844.         284910.
## 18 Menores de 14 04              698172.         307543.
## 19 Menores de 14 07             1126737.         409562.
## 20 Menores de 14 10             1195499          273635.
```

Tip:

Este código usa `substr(indice_tiempo, 6, 7)` para definir el mes. Esta línea lo que hace es quedarse con el texto de `indice_tiempo` que está entre la posición 6 y la 7. De esta manera, del texto `"2012-01"` se queda con "`01`", que representa el mes.

Esta no es la mejor manera de trabajar con fechas, pero es suficiente por ahora.

El resultado va a siempre ser una tabla con la misma cantidad de filas que grupos y una cantidad de columnas igual a la cantidad de columnas usadas para agrupar y los estadísticos computados.

Desafío:

¿Cuál te imaginás que va a ser el resultado del siguiente código? ¿Cuántas filas y columnas va a tener? (Tratá de pensarlo antes de correrlo.)

```
turistas_edad %>%
   summarise(turistas_promedio = mean(turistas))
```

El combo `group_by()%>% summarise()` se puede resumir en esta figura. Las filas de una tabla se dividen en grupos, y luego cada grupo se "resume" en una fila en función del estadístico usado.

Al igual que hicimos "cuentas" usando algunas variables numéricas para obtener información nueva, también podemos utilizar variables categóricas. No tiene sentido calcular `mean(edad)` ya que en esta tabla la edad está codificada como texto, pero tal vez te interese *contar* la cantidad de observaciones por edad:

```
turistas_edad %>%
  group_by(edad) %>%
  summarise(cantidad = n())
```

```
## # A tibble: 5 x 2
##   edad          cantidad
##   <chr>            <int>
## 1 14 a 29             36
## 2 30 a 44             36
## 3 45 a 59             36
## 4 60 o mas            36
## 5 Menores de 14       36
```

En este caso se ve que hay 36 observaciones (36 fechas) para todos los rangos etáreos.

5.4. Creando nuevas columnas con `mutate()`

Todo esto está bien para hacer cálculos con columnas previamente existentes, pero muchas veces tenés que crear nuevas columnas.

La tabla `turistas_edad` tiene información temporal codificada como texto en el formato "año-mes". Sería mucho mejor que esté codificada como una fecha. Una forma muy simple de convertir caracteres a fechas es usando el paquete **lubridate**. Este paquete tiene un montón de funciones que te facilitan la vida al trabajar con fechas, pero en este caso vamos a usar la función `ym()` que transforma en fecha cualquier texto que tenga una fecha codificada con el año seguido del mes.

Para agregar una columna llamada `tiempo` a la tabla `turistas_edad`, vamos a usar la función `mutate()`

```
turistas_edad %>%
  mutate(tiempo = lubridate::ym(indice_tiempo))
```

```
## # A tibble: 180 x 4
##    indice_tiempo edad          turistas tiempo
##    <chr>         <chr>            <dbl> <date>
##  1 2012-01       Menores de 14  2700561 2012-01-01
##  2 2012-01       14 a 29        3143845 2012-01-01
##  3 2012-01       30 a 44        3234308 2012-01-01
##  4 2012-01       45 a 59        2126707 2012-01-01
##  5 2012-01       60 o mas       1268445 2012-01-01
##  6 2012-04       Menores de 14   957231 2012-04-01
##  7 2012-04       14 a 29        1296808 2012-04-01
##  8 2012-04       30 a 44        1377356 2012-04-01
##  9 2012-04       45 a 59        1021227 2012-04-01
## 10 2012-04       60 o mas        771842 2012-04-01
## # ... with 170 more rows
```

¿Notás la diferencia entre la columna `indice_tiempo` y `tiempo`?

Recordá que las funciones de dplyr nunca modifican la tabla original. `mutate()` devolvió una nueva tabla que es igual a la tabla `turistas_edad` pero con la columna "tiempo" agregada. La tabla `turistas_edad` sigue intacta.

Tip:

Para usar la función `ym()` del paquete lubridate el código de arriba usa `lubridate::ym()`. Esta es una forma de usar funciones de paquetes sin tener que cargarlos con `library()`.

5.5. Desagrupando con `ungroup()`

En general, la mayoría de las funciones de dplyr "entienden" cuando una tabla está agrupada y realizan las operaciones para cada grupo.

Desafío:

¿Cuál de estos dos códigos agrega una columna llamada "turistas_promedio" con la cantidad de turistas promedios para cada rango de dades? ¿Qué hace el otro?

```
turistas_edad %>%
  group_by(edad) %>%
  mutate(turistas_promedio = mean(turistas))

turistas_edad %>%
  mutate(turistas_promedio = mean(turistas))
```

En otras palabras, los resultados de `mutate()`, `filter()`, `summarise()` y otras funciones cambian según si la tabla está agrupada o no. Como a veces uno se puede olvidar que quedaron grupos, es conveniente usar la función `ungroup()` una vez que dejás de trabajar con grupos:

```
turistas_edad %>%
  group_by(edad) %>%
  mutate(turistas_promedio = mean(turistas)) %>%
  ungroup()
```

```
## # A tibble: 180 x 4
##    indice_tiempo edad          turistas turistas_promedio
##    <chr>         <chr>            <dbl>             <dbl>
##  1 2012-01       Menores de 14  2700561          1368313.
##  2 2012-01       14 a 29        3143845          1440779.
##  3 2012-01       30 a 44        3234308          1530064.
##  4 2012-01       45 a 59        2126707          1242697.
##  5 2012-01       60 o mas       1268445          1033486.
##  6 2012-04       Menores de 14   957231          1368313.
##  7 2012-04       14 a 29        1296808          1440779.
##  8 2012-04       30 a 44        1377356          1530064.
##  9 2012-04       45 a 59        1021227          1242697.
## 10 2012-04       60 o mas        771842          1033486.
## # ... with 170 more rows
```

Capítulo 6

Gráficos con ggplot2

Visualizar datos es útil para identificar a relación entre distintas variables pero también para comunicar el análisis de los datos y resultados. El paquete **ggplot2** permite generar gráficos de gran calidad en pocos pasos. Cualquier gráfico de ggplot tendrá como mínimo 3 componentes: los **datos**, un **sistema de coordenadas** y una **geometría** (la representación visual de los datos) y se irá construyendo por capas.

6.1. Primera capa: el área del gráfico

Cómo siempre, primero hay que cargar los paquetes y los datos. Para esta sección, vamos a leer el archivo parques_tidy.csv, que es una serie de tiempo de visitantes a parques nacionales descargada de Yvera y modificada un poco para poder trabajar con ella más fácilmente con ggplot2.

```
library(ggplot2)
parques <- readr::read_csv("datos/parques_tidy.csv")
```

Para tener una idea de este conjunto de datos, `head()` muestra las primeras 6 filas

```
head(parques)
```

```
## # A tibble: 6 x 5
##   indice_tiempo region       residentes noresidentes  total
##   <date>        <chr>             <dbl>        <dbl>  <dbl>
## 1 2008-01-01    buenos-aires        885            0    885
## 2 2008-01-01    cordoba             717          145    862
```

```
## 3 2008-01-01    cuyo              4965         179   5144
## 4 2008-01-01    litoral         111408       55335 166743
## 5 2008-01-01    norte             4241         774   5016
## 6 2008-01-01    patagonia       230087      141973 372060
```

La tabla tiene

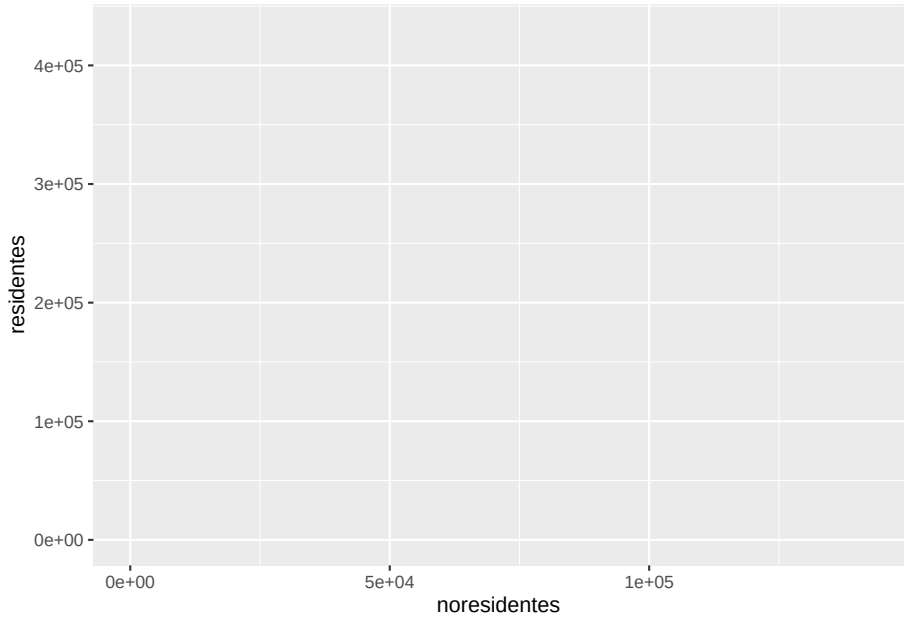
- **indice_tiempo**: fecha que representa el mes,
- **region**: texto con la región,
- **residentes**: cantidad de visitantes residentes que visitaron cada región en cada fecha,
- **noresidentes**: cantidad de visitantes no residentes,
- **total**: cantidad todal de visitantes (la suma de `residentes` y `noresidentes`)

La función principal de ggplot2 es justamente `ggplot()` que permite *iniciar* el gráfico y además definir las características *globales*. El primer argumento de esta función serán los datos que vas a visualizar, siempre en un data frame. En este caso usamos `parques`.

El segundo argumento se llama "mapping" (*mapeo* en inglés). Este argumento define la relación entre cada columna del data frame y los distintos parámetros gráficos. Por ejemplo, qué columna va a representar el eje x, cuál va a ser el eje y, etc.. Este mapeo se hace **siempre** con la función `aes()` (que viene de *aesthetics*, *estética* en inglés).

Por ejemplo, si querés hacer un gráfico que muestre la relación entre la cantidad de visitantes residentes y no residentes usarías algo como esto:

```
ggplot(data = parques, mapping = aes(x = noresidentes, y = residentes))
```

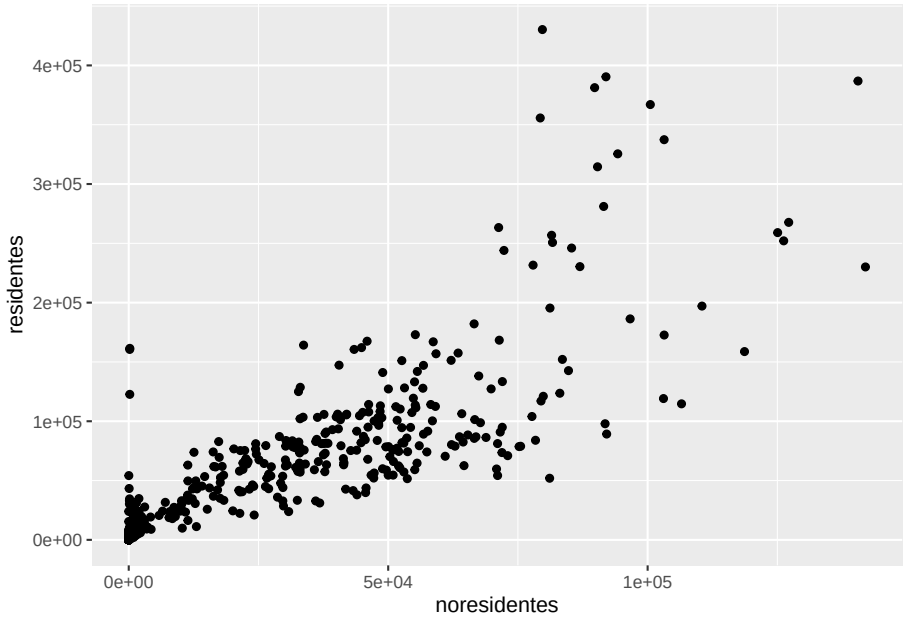


Este código le indica a ggplot que genere un gráfico donde el eje **x** se mapea a la columna `noresidentes` y el eje **y**, a la columna `residentes`. Pero, como se ve, esto sólo genera el área del gráfico y los ejes. Lo que falta es indicar con qué geometrías representar los datos.

6.2. Segunda capa: geometrías

Para agregar geometrías que representen los datos lo que hay que hacer es *sumar* el resultado de una función que devuelva una capa de geometrías. Estas suelen ser funciones que empiezan con "geom_" y luego el nombre de la geometría (en inglés). Para representar los datos usando puntos, hay que uasr `geom_point()`

```
ggplot(data = parques, mapping = aes(x = noresidentes, y = residentes)) +
  geom_point()
```



¡Tu primer gráfico!

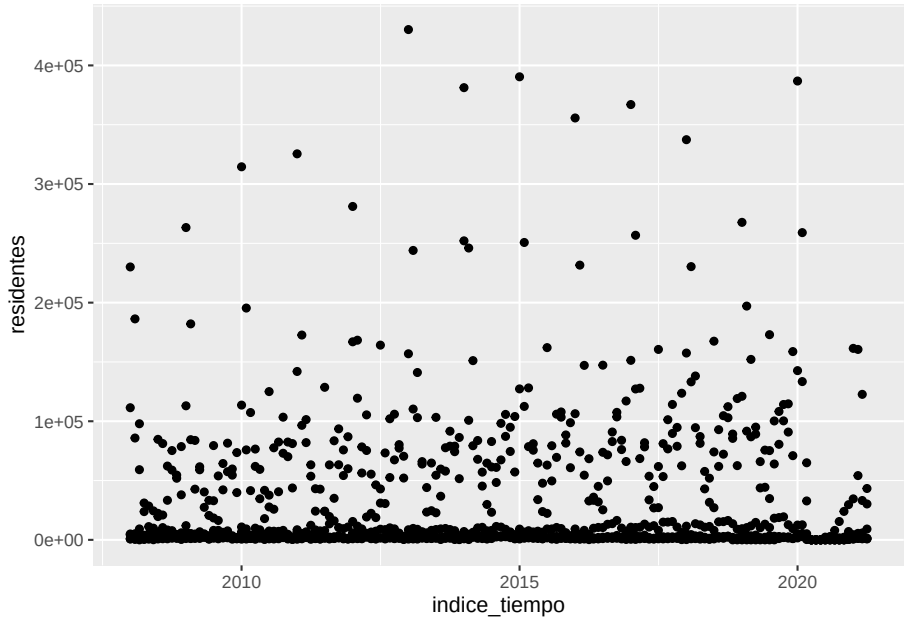
Primer desafío

Ahora es tu turno. Modifica el gráfico anterior para visualizar cómo cambia la cantidad de visitantes residentes a lo largo de los años.

¿Te parece útil este gráfico?

Este gráfico tiene un punto por cada región y cada trimestre, pero es posible identificar a qué región corresponde cada punto. Es necesario agregar información al gráfico.

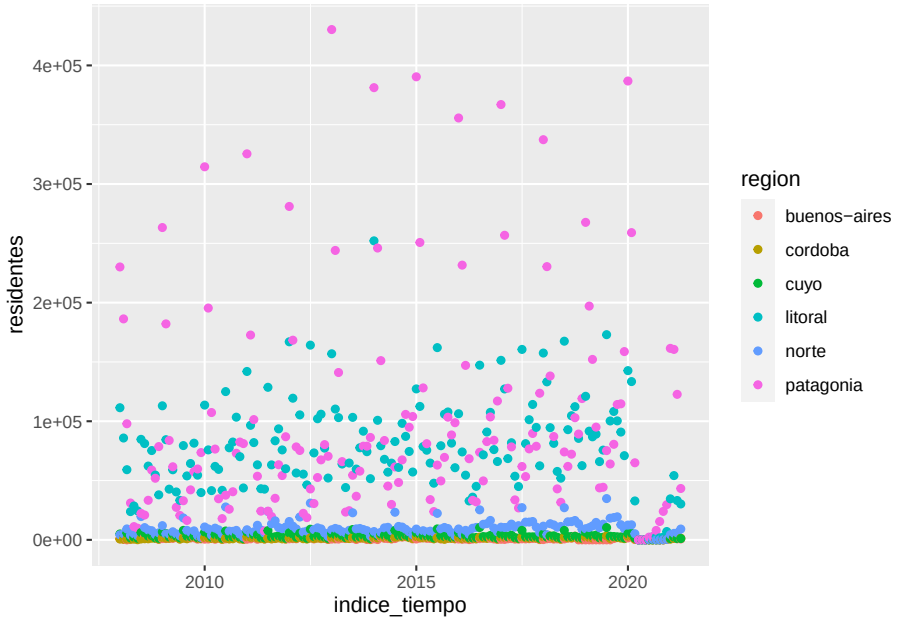
```
ggplot(data = parques, mapping = aes(x = indice_tiempo, y = residentes)) +
  geom_point()
```



6.3. Mapear variables a elementos

Una posible solución sería utilizar otras variables de los datos, por ejemplo `region` y *mapear* el color de los puntos de a cuerdo a la región a la que pertenecen.

```
ggplot(data = parques, mapping = aes(x = indice_tiempo, y = residentes)) +
  geom_point(aes(color = region))
```



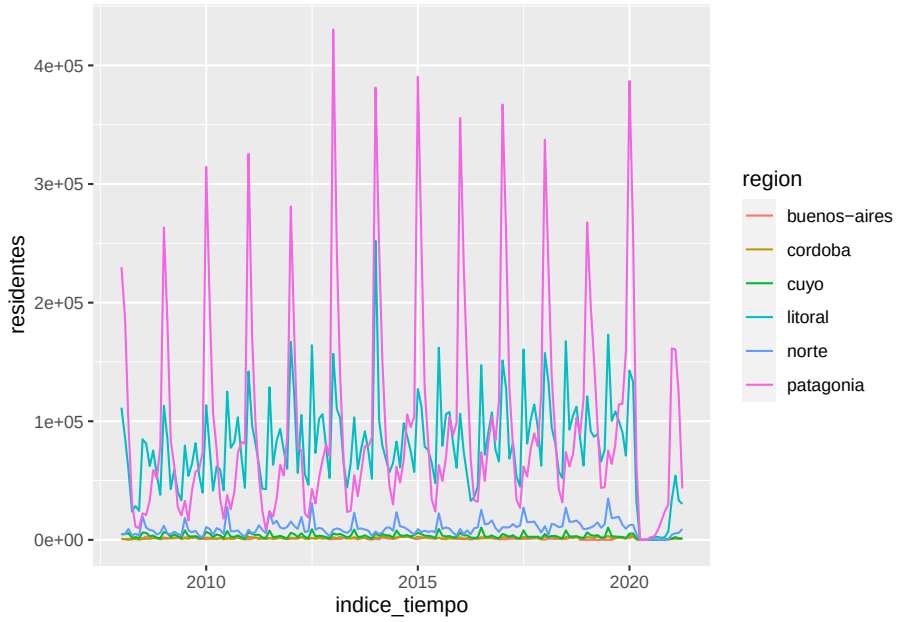
Ahh, ahora está un poco mejor. Se puede ver que la Patagonia (los puntos fucsia) tiene más visitantes que el resto de las regiones. Le sigue la región del Litoral (puntos turquesa).

Algo muy importante a tener en cuenta: **los puntos toman un color de acuerdo a una variable de los datos**, y para que ggplot2 identifique esa variable (en este caso `region`) es necesario incluirla dentro de una función `aes()`.

6.4. Otras geometrías

Este gráfico posiblemente no sea muy adecuado si queremos visualizar la *evolución* de una variable a lo largo del tiempo. Si bien se pueden identificar a qué región correponde cada punto, es muy difícil seguir la evolución de uno en particular; especialmente en la Patagonia, donde hay mucha distancia vertical entre los puntos. Lo más natural es cambiar la geometría a lineas usando `geom_line()`

```
ggplot(data = parques, mapping = aes(x = indice_tiempo, y = residentes)) +
  geom_line(aes(color = region))
```



Por suerte las funciones `geom_*()` tienen más o menos nombres amigables.

Y ahora si, conseguimos el gráfico que estamos buscando. Las líneas unen puntos consecutivos y permiten que el ojo siga la evolución de cada región. La diferencia entre temporada alta y temporada baja en Patagonia (la estacionalidad) salta inmediatamente.

Segundo desafío

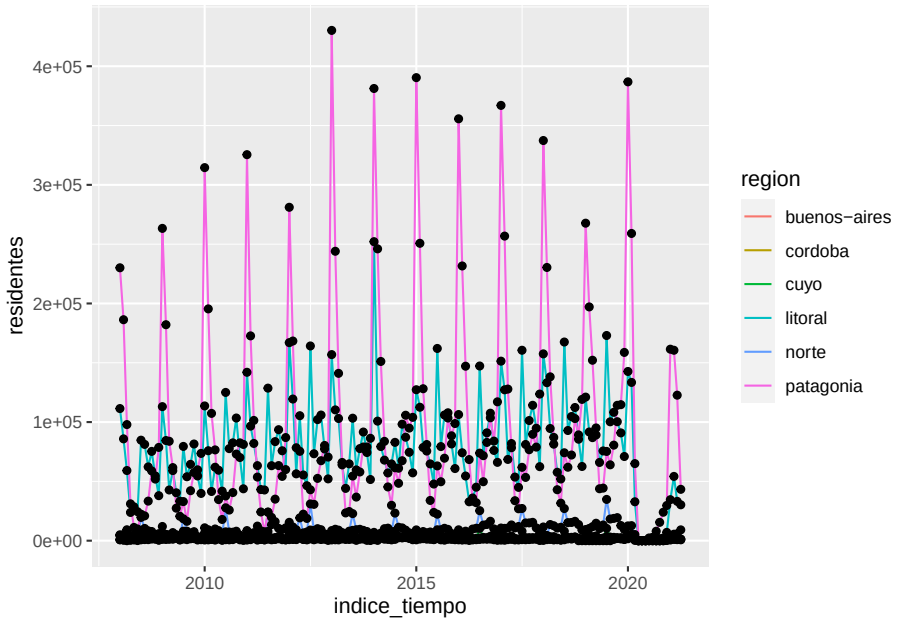
Hasta ahora tenemos dos capas: el área del gráfico y una única geometría (las líneas).

1. Sumá una tercera capa para visualizar puntos además de las líneas.
2. ¿Porqué los puntos ahora no siguen los colores de las regiones?
3. ¿Qué cambio podrías hacer para que los puntos también tengan color según la región?

Acá surge una característica importante de las capas: pueden tener apariencia independiente si solo *mapeamos* el color en la capa de las líneas y no en la capa de los puntos. Al mismo tiempo, si quisiéramos que todas las capas tenga la misma apariencia podemos incluir el argumento `color =`en la función global `ggpplot()` o repetirlo en cada capa.

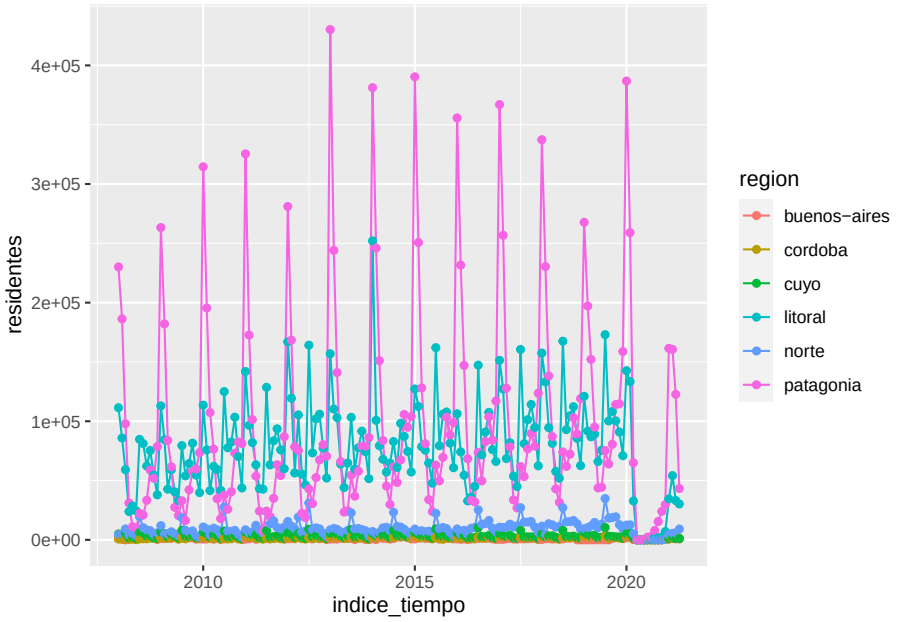
Es la diferencia entre esto

```
ggplot(parques, aes(indice_tiempo, residentes)) +
  geom_line(aes(color = region)) +
  geom_point()
```



y esto.

```
ggplot(parques, aes(indice_tiempo, residentes, color = region)) +
  geom_line() +
  geom_point()
```



Si te preguntás a donde fueron a parar el `data =`, el `mapping =` y los nombres de los argumentos adentro de la función `aes()`, `x =` e `y =`, resulta que estamos aprovechando que tanto ggplot2 como nosotros ahora sabemos en que orden recibe la información cada función. Siempre el primer elemento que le *pases* o indiquemos a la función `ggplot()` será el data frame y el segundo será el `aes()`.

Algunos argumentos para cambiar la apariencia de las geometrías son:

- `color` o `colour` modifica el color de líneas y puntos
- `fill` modifica el color *interno* de un elemento, por ejemplo el relleno de una barra
- `linetype` modifica el tipo de línea (punteada, continua, con guiones, etc…)
- `size` modifica el tamaño de los elementos (por ejemplo el tamaño de puntos o el grosor de líneas)
- `alpha` modifica la transparencia de los elementos (1 = opaco, 0 = transparente)
- `shape` modifica el tipo de punto (círculos, cuadrados, triángulos, etc.)

El *mapeo* entre una variable y un parámetro de geometría se hace a través de una **escala**. La escala de colores es lo que define, por ejemplo, que los puntos donde la variable `region` toma el valor `"patagonia"` van a tener el color rosa (●), donde toma el valor `"córdoba"`, mostaza (●), etc…

Modificar elementos utilizando un valor único

Es posible que en algún momento necesites cambiar la apariencia de los elementos o geometrías independientemente de las variables de tu data frame. Por ejemplo podrías querer que todos los puntos sean de un único color: rojos. En este caso `geom_point(aes(color = "red"))` no va a funcionar -ojo que los colores van en inglés-. Lo que ese código hace es mapear el parámetro geométrico "color" a una variable que contiene el valor `"red"` para todas las filas. El mapeo se hace a través de la escala, que va a asignarle un valor (rosa) a los puntos correspondientes al valor `"red"`.

Como en este caso no te interesa *mapear* el color a una variable, tenés que mover ese argumento **afuera** de la función `aes()`: `geom_point(color = "red")`.

6.5. Relación entre variables

Muchas veces no es suficiente con mirar los datos crudos para identificar la relación entre las variables; es necesario usar alguna transformación estadística que resalte esas relaciones, ya sea ajustando una recta o calculando promedios.

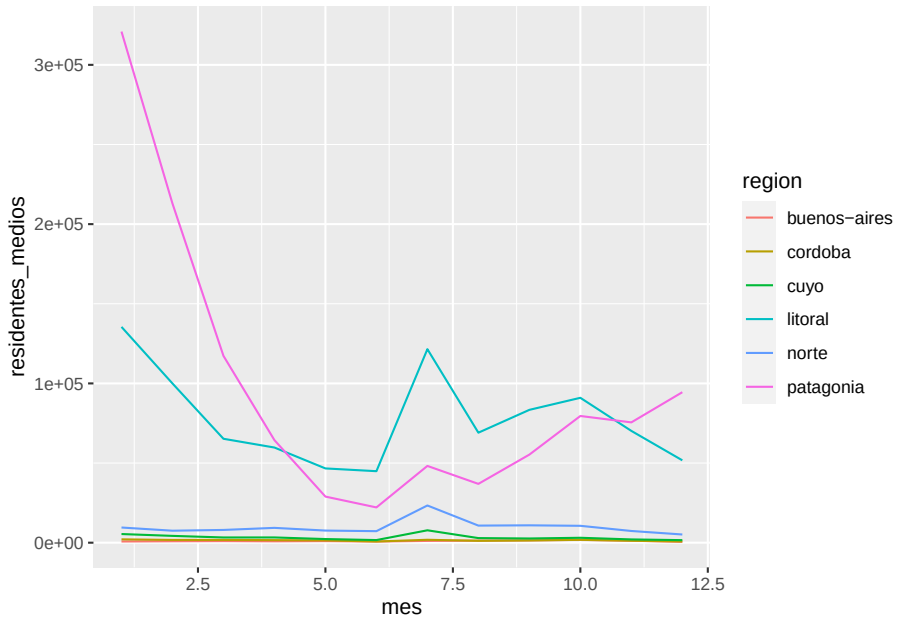
Para alguna transformaciones estadísticas comunes, ggplot2 tiene geoms ya programados, pero muchas veces es posible que necesites manipular los datos antes de poder hacer un gráfico. A veces esa manipulación será compleja y su resultado luego va a ser utilizado en otras partes del análisis. En esos casos, te conviene guardar los datos modificados en una nueva variable. Pero para transformaciones más simples podés *encadenar* la manipulación de los datos directamente en el gráfico.

Por ejemplo, en un gráfico anterior viste que hay un ciclo estacional bastante notorio en la cantidad de visitantes. Para visualizar el ciclo anual medio de toda la serie podés calcular la cantidad promedio de visitantes por cada mes y región usando dplyr y luego graficar eso:

```
library(dplyr)

parques %>%
  group_by(mes = lubridate::month(indice_tiempo), region) %>%
  summarise(residentes_medios = mean(residentes)) %>%
  ggplot(aes(mes, residentes_medios)) +
  geom_line(aes(color = region))
```

```
## `summarise()` has grouped output by 'mes'. You can override using the `.groups` argu
```

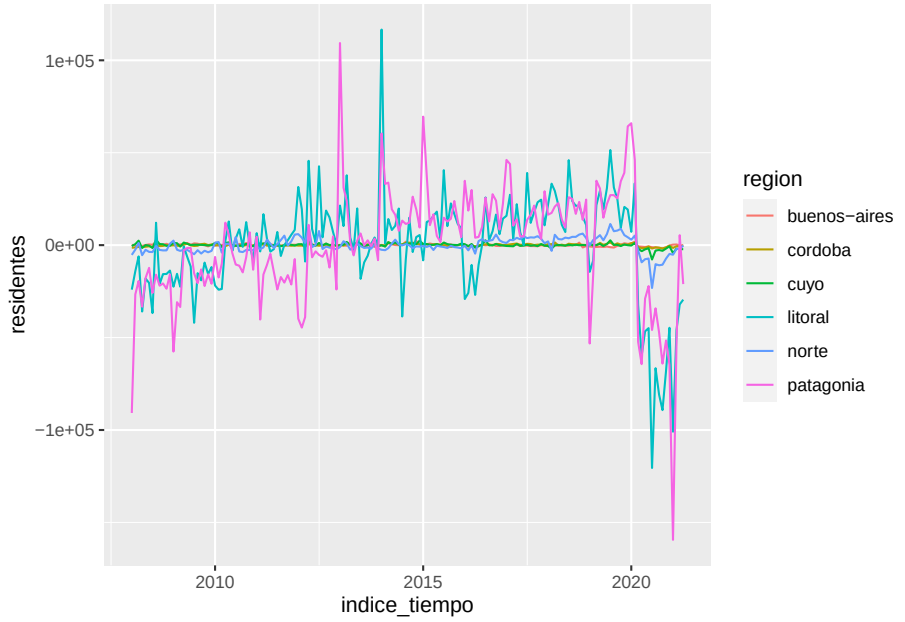


Esto es posible gracias al operador `%>%` que le *pasa* el resultado de `summarise()` a la función `ggplot()`. Y este resultado no es ni más ni menos que el data frame que necesitás para hacer el gráfico. Es importante notar que una vez que comenzamos el gráfico ya **no** se puede usar el operador `%>%` y las capas del gráfico se *suman* como siempre con `+`.

El gráfico muestra que en un enero típico, los parques de la región Patagonia esperan algo más de 300.000 visitantes residentes, mientras que en junio tienen menos de 25.000. La cantidad de visitas de residentes a los parques del litoral es un poco más constante a lo largo del año.

Una vez analizado el ciclo anual, podrías querer filtrarlo de los datos para obtener una serie *desestacionada*. Una forma de hacerlo es restando la media así:

```
parques %>%
  group_by(trimestre = lubridate::month(indice_tiempo), region) %>%
  mutate(residentes = residentes - mean(residentes)) %>%
  ggplot(aes(indice_tiempo, residentes)) +
  geom_line(aes(color = region))
```

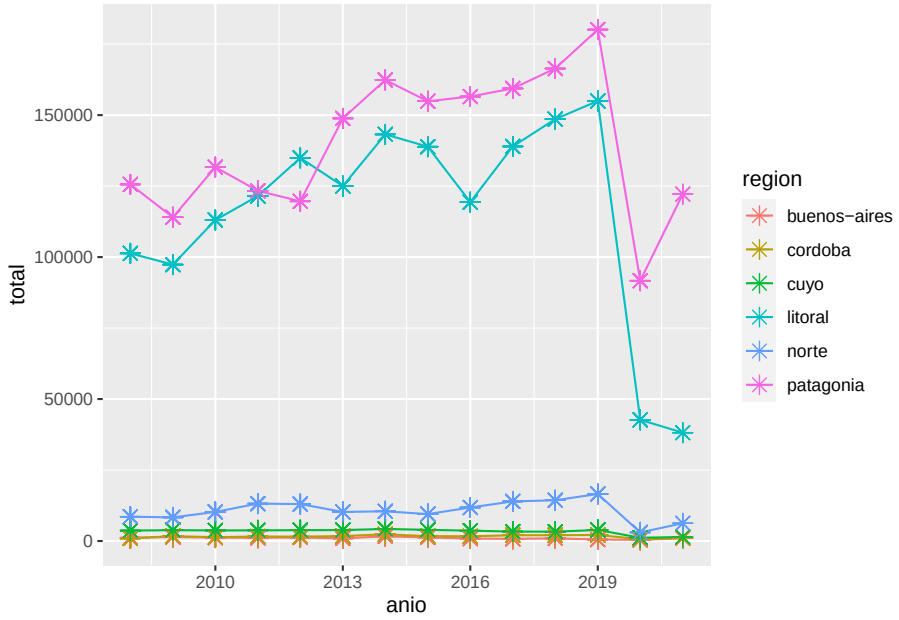


Al filtra el ciclo anual medio, saltan a la vista otros patrones de variabilidad. Se puede ver que tanto en la Patagonia como en el Litoral la cantidad de visitantes residentes estuvo aumentando hasta 2020, cuando a causa de la pandemia, la visitas se desplomaron. También se pueden destacar meses interesantes donde los parques recibieron muchos más visitantes de lo que es normal para ese mes.

Tercer desafío

Modificá el siguiente código para obtener el gráfico que se muestra más abajo.

```
parques %>%
  mutate(anio = lubridate::year(indice_tiempo)) %>%
  group_by(region, ____) %>%
  mutate(total = mean(total)) %>%
  ggplot(aes(anio, ___)) +
  geom_line(aes(color = region)) +
  geom_point(aes(color = region), shape = ____, size = 3)
```



6.6. Transformaciones estadísticas

Hasta ahora visualizamos los datos tal cual vienen en la base de datos o transformados con ayuda de dplyr, pero hay ciertas transformaciones comunes que se pueden hacer usando ggplot2.

Para esta sección vamos a usar la tabla de microdatos de la Encuesta de Viajes y Turismo de los Hogares (EVYTH). Vamos a seleccionar sólo las columnas que identifican cada hogar y viaje, y el gasto del viaje y el quintil de ingreso del hogar. Como cada fila es una persona y cada hogar puede tener más de una persona, el código de abajo usa `distinct()` para eliminar los valores repetidos.

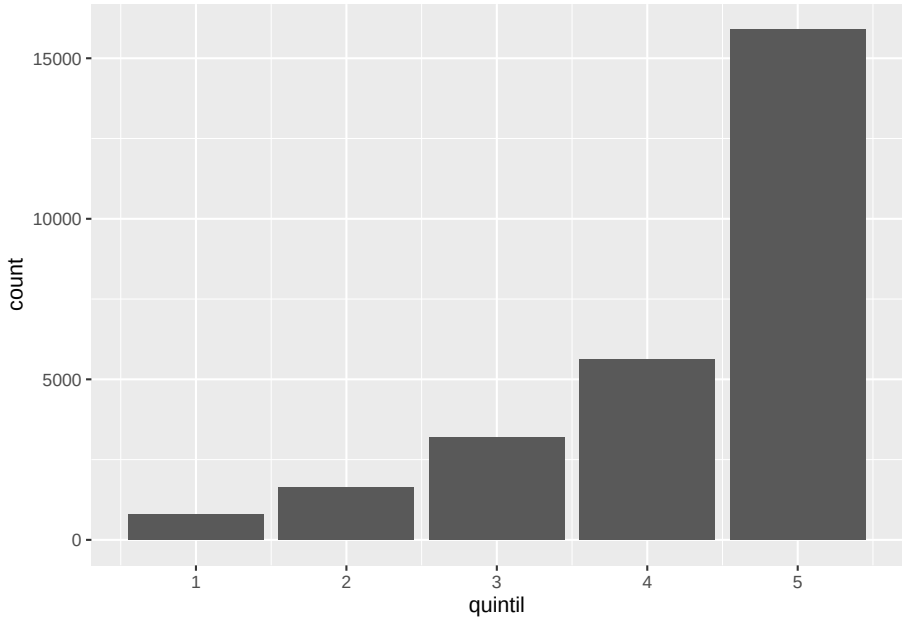
```
gastos <- readr::read_csv("datos/evyth_2019_2021t1.csv") %>%
  select(id_hogar, id_viajes, anio, trimestre,
         region_destino, gasto = gasto_pc, quintil = quintil_pcf_visitante) %>%
  distinct() %>%
  mutate(region_destino = factor(region_destino))
```

6.7. Gráficos de frecuencias

Este es un gráfico de barras construido usando la función `geom_bar()`. En el eje x muestra el `quintil` de cada hogar y en el eje y la cantidad (*count* en inglés)

de hogares en ese quintil. Habrás notado que el data frame `gastos` no tiene ninguna variable que se llame `count` y en ninguna parte del código se calcula esa cantidad explícitamente. Esta variable es computada por `geom_bar()`.

```
ggplot(gastos, aes(quintil)) +
  geom_bar()
```

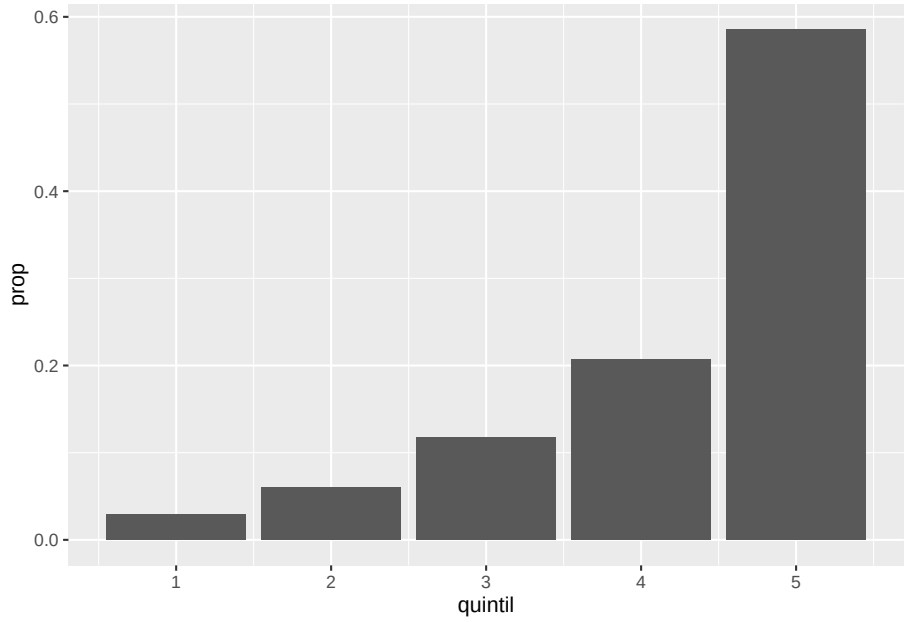


Tercer desafío

¿Qué otra variable, además de `count` computa `geom_bar()`? Andá a la documentación de `geom_bar()` (apretando F1 sobre el nombre de la función o ejecutando `?geom_bar` en la consola) y andá a la sección llamada "Computed variables" para verlo.

Además de contar la cantidad de elementos, `geom_bar()` computa la proporción sobre el total que representa grada grupo con la variable computada `prop`. Para usar esa variable computada como la altura de las barras hay que usar la función `stat()` dentro del `aes()`.

```
ggplot(gastos, aes(quintil)) +
  geom_bar(aes(y = stat(prop)))
```

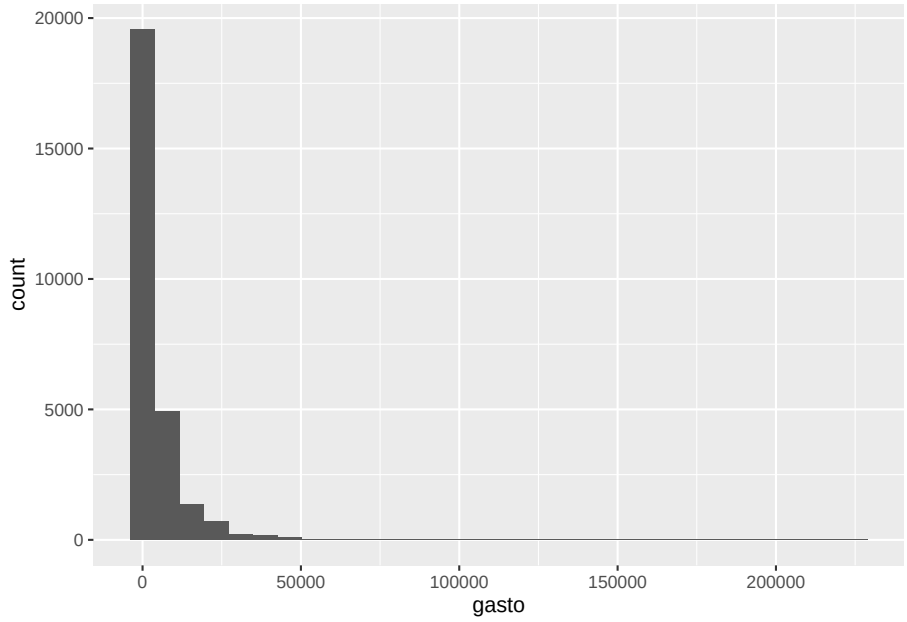


Ahora podés ver que casi el 60 % de las familias encuestadas pertenecen al quintil 5.

Para obtener algo parecido pero para variables continuas hay que usar `geom_histogram()`. Un ejemplo de variable continua es el `gasto` asociado a cada viaje.

```
ggplot(gastos, aes(gasto)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

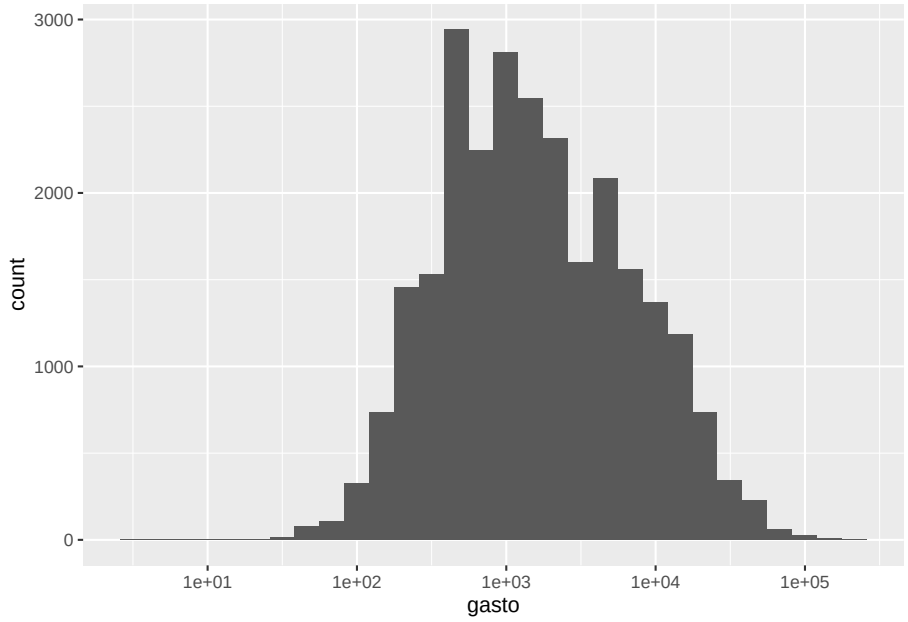


Algo que sucede muy seguido con medidas de gastos e ingresos, es que esta variable tiene una distribución altamente asimétrica. Es decir, hay muchos viajes con valores muy bajos y muy pocos con valores muy altos. Esto hace que se pierda detalle en el rango de valores donde se encuentra la mayoría de las observaciones. Una forma de resolver esto es transformando los valores con el logaritmo, pero graficar el logaritmo del gasto no sería muy fácil de interpretar. En vez de eso, es recomendable utilizar una transformación de escala en el gráfico.

Para transformar los valores del eje x con el logaritmo, se usa `scale_x_log10()`. Esto realiza la transformación pero luego muestra las etiquetas en la escala original.

```
ggplot(gastos, aes(gasto)) +
  geom_histogram() +
  scale_x_log10()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Primer desafío

¿Notaste el mensaje que devuelve el gráfico?

‘`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.‘

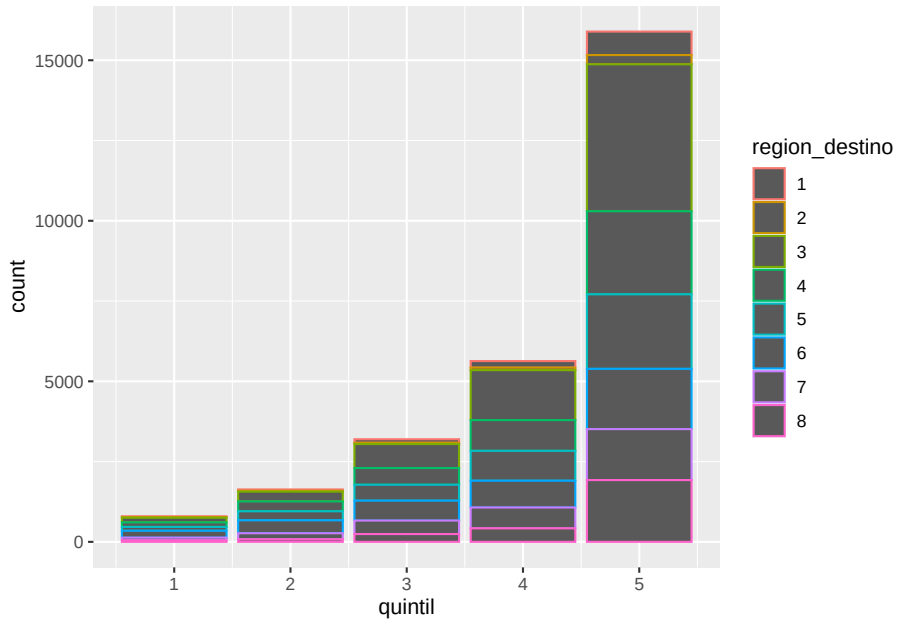
Esta geometría tiene dos argumentos importantes `bins` y `binwidth`. Cambiá el valor de alguno de los dos argumentos y volvé a generar el gráfico, ¿que rol juegan los argumentos?

También podés revisar la documentación.

6.7.1. Posición

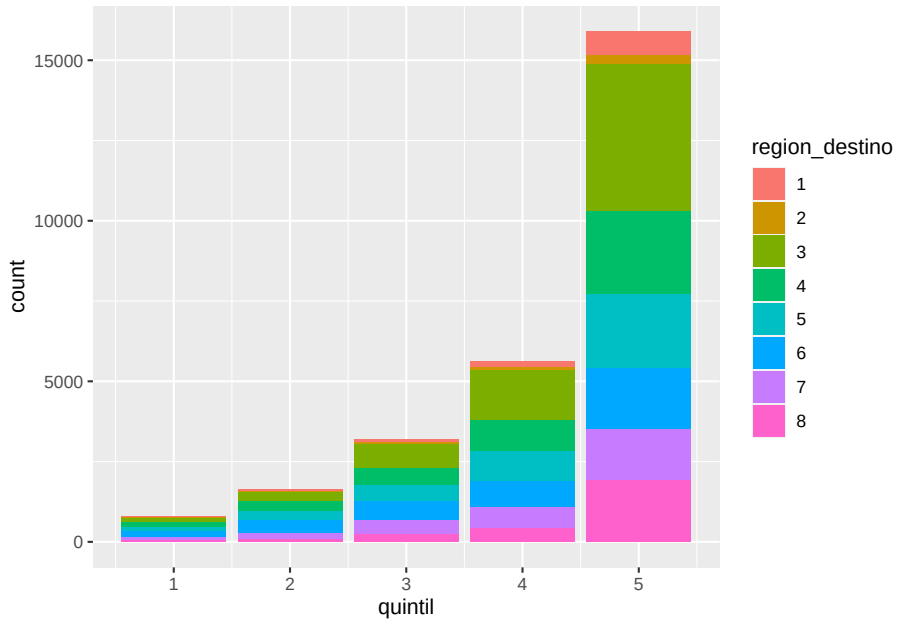
Es posible que la distribución en quintiles de los visitantes no sea igual para cada región, entonces podrías querer dibujar una barra para cada quintil y cada región e identificar cada quintil con un color disitnto. Siguiendo lo anterior, quizás lo primero que se te ocurre es algo como esto:

```
ggplot(gastos) +
  geom_bar(aes(quintil, color = region_destino))
```



El problema de esto es que el parámetro "color" de las barras define el color del contorno, no el relleno. Para modificar el relleno hay que cambiar el parámetro `fill`.

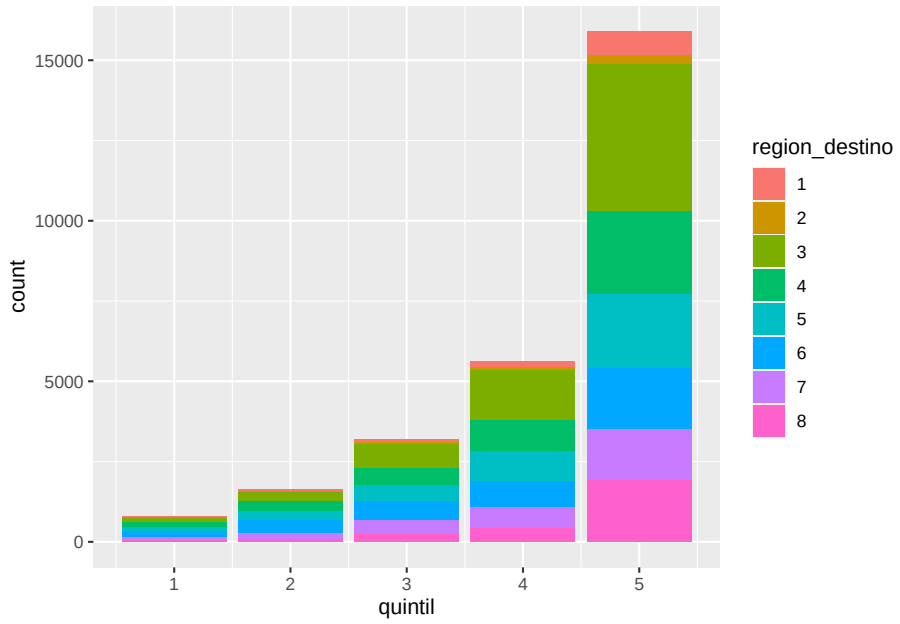
```
ggplot(gastos) +
  geom_bar(aes(quintil, fill = region_destino))
```



Al *mapear* una variable distinta, se puede visualizar información extra. En el gráfico, cada barra está compuesta de 8 barras apiladas cuya altura representa la cantidad de hogares que viajaron a cada región y están en cada cuantil. Este "apilamiento" de las barras es la opción de posición por defecto, pero puede cambiarse con el argumento "position".

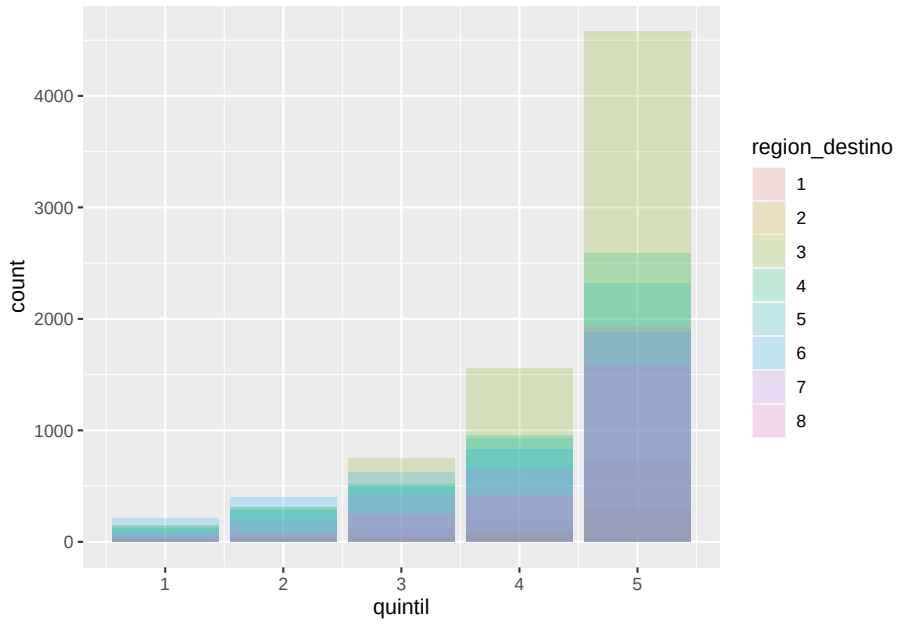
La posición por defecto es "stack".

```
ggplot(gastos) +
  geom_bar(aes(quintil, fill = region_destino), position = "stack")
```



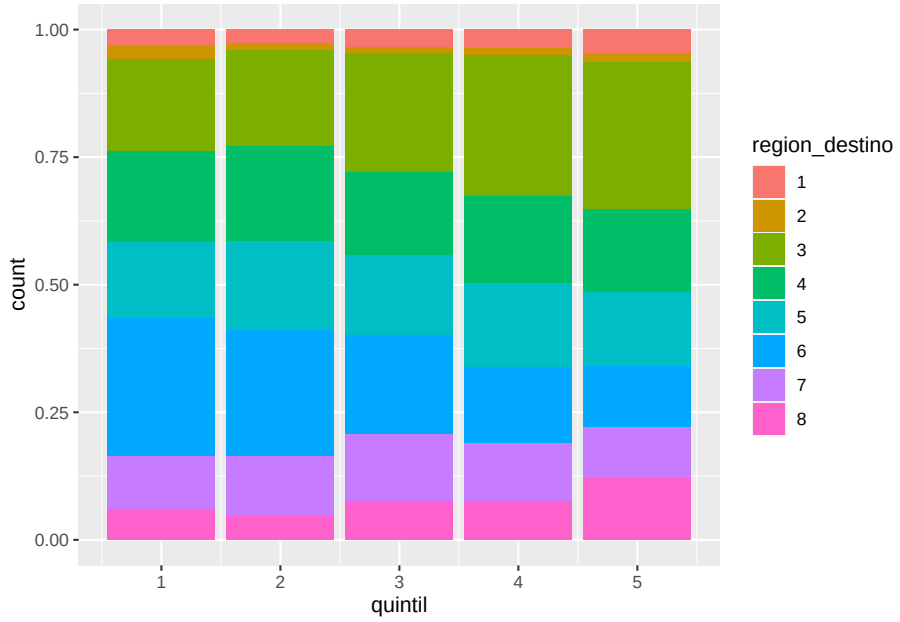
`position = "identity"` colocará cada barra comenzando en cero quedando todas superpuestas. Para ver esa superposición, debemos hacer que las barras sean ligeramente transparentes configurando el `alpha` a un valor pequeño.

```
ggplot(gastos) +
  geom_bar(aes(quintil, fill = region_destino), alpha = 0.2, position = "identity")
```



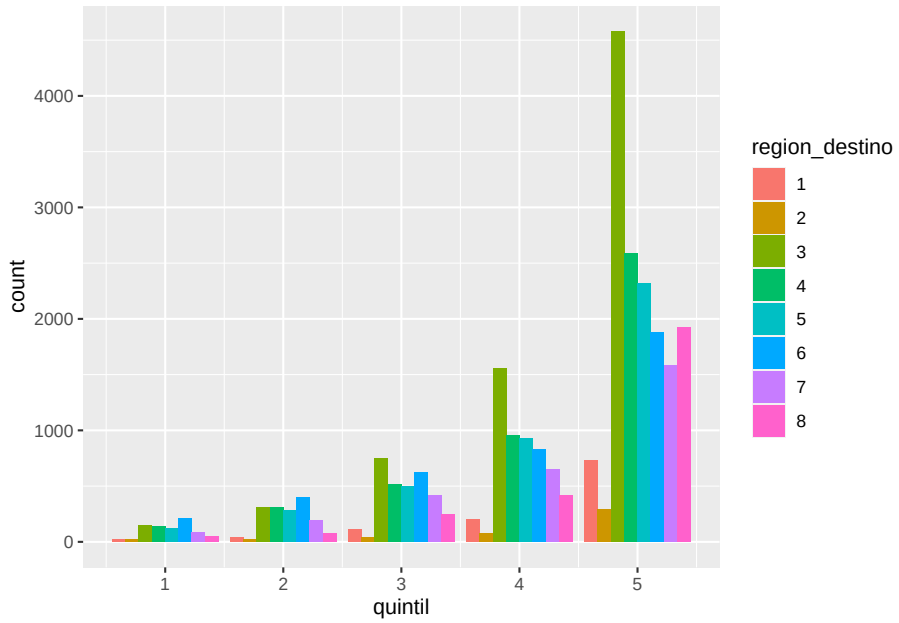
`position = "fill"` apila las barras al igual que `position = "stack"`, pero transforma los datos para que cada conjunto de barras apiladas tenga la misma altura. Esto hace que sea más fácil comparar proporciones entre grupos.

```
ggplot(gastos) +
  geom_bar(aes(quintil, fill = region_destino),  position = "fill")
```



`position = "dodge"` coloca las barras una al lado de la otra. Esto hace que sea más fácil comparar valores individuales.

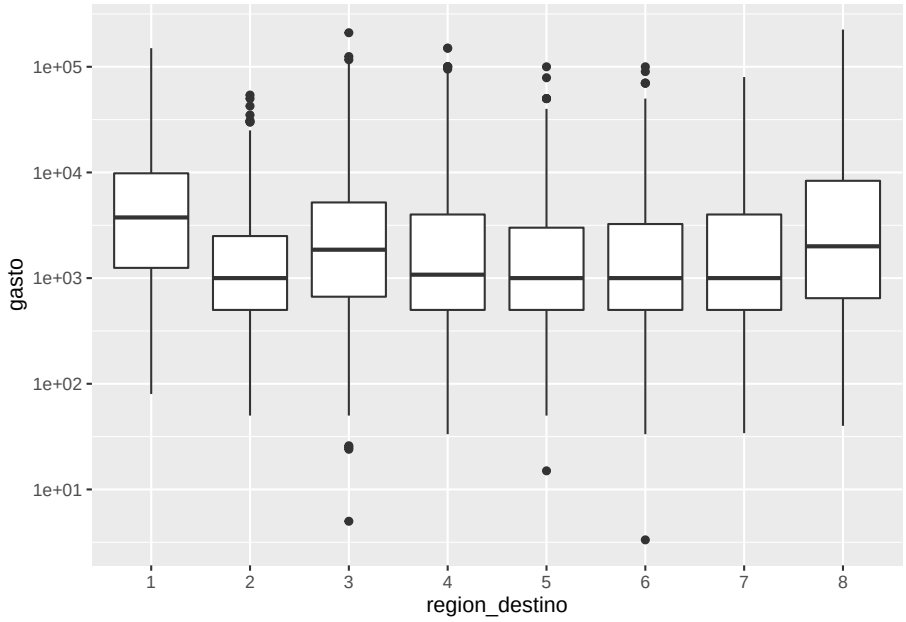
```
ggplot(gastos) +
  geom_bar(aes(quintil, fill = region_destino),  position = "dodge")
```



6.8. Gráficos de caja

Los diagramas de caja –mejor conocidos como boxplots– calculan un resumen de los valores centrales y de dispersión de la distribución de los datos.

```
ggplot(gastos, aes(region_destino, gasto)) +
  geom_boxplot() +
  scale_y_log10()
```



La línea central de la caja corresponde a la **mediana** (el valor que toma el dato central) y los extremos de la caja son los **cuartiles 1 y 3**, definiendo así el **rango intercuartil** (IQR). Los extremos están definidos como el valor observado que no esté más lejos de **1.5*IQR** de la mediana y los puntos son las observaciones que se escapan de ese rango, que pueden ser considerados **outliers** o **valores extremos**.

Los boxplot brindan algo de información sobre la distribución de los datos pero al mismo tiempo *esconden* la forma de la distribución y el número de datos que se usaron para generarlos. Por esta razón también existen `geom_violin()` y `geom_jitter()`.

Segundo desafío

1. Volvé a graficar la distribución del precio para cada tipo de claridad pero ahora usando `geom_violin()` y `geom_jitter()`.
2. ¿Qué ventajas y desventajas encuentran respecto de `geom_boxplot()`?

```
ggplot(gastos, aes(region_destino, gasto)) +
  geom_violin() +
  scale_y_log10()
```

```
ggplot(gastos, aes(region_destino, gasto)) +
  geom_jitter(alpha = 0.2, size = 0.1) +
  scale_y_log10()
```

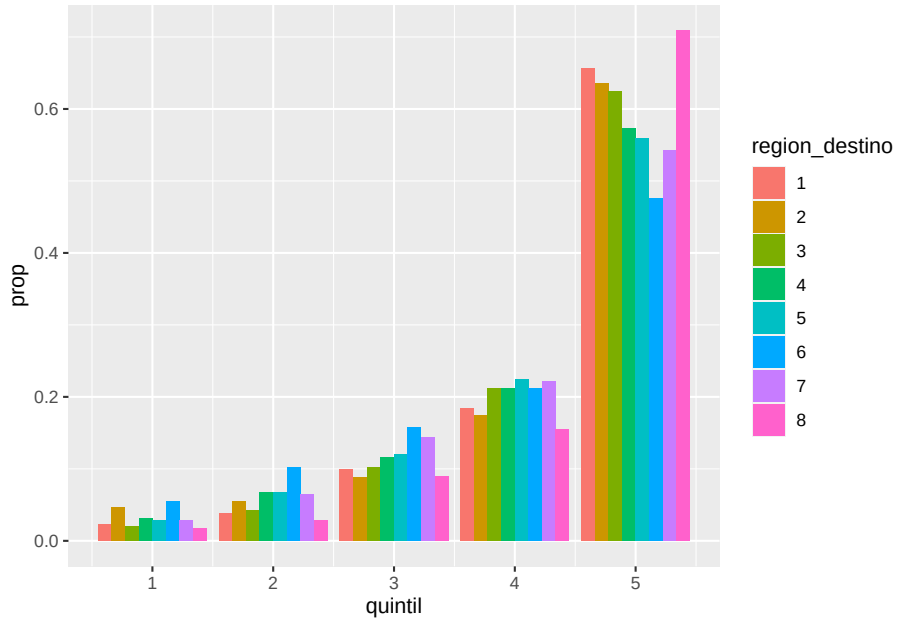
Cuando nuestra base de datos es muy grande corremos el riesgo de generar de que los elementos del gráfico estén tan juntos que se solapen y no se vean. Esto se conoce como **overplotting**. La tabla `gastos` tiene 27149 observaciones y al graficar un punto por cada una, aún si están separados por la región, quedan superpuestos.

Por esto es que en el último gráfico los puntos son muy chiquitos y con transparencia.

6.9. Graficando en múltiples paneles

En un gráfico anterior mostramos la cantidad de hogares en cada quintil en función de la región de destino mapeando la variable `region_destino` al relleno de las columnas:

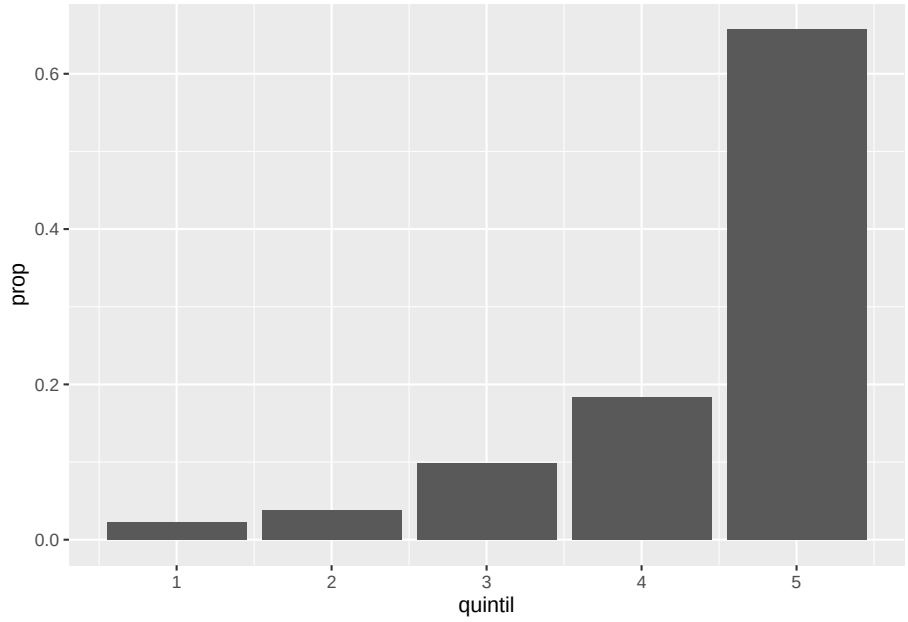
```
ggplot(gastos) +
  geom_bar(aes(quintil, y = stat(prop), fill = region_destino),  position = "dodge")
```



Este gráfico permite comparar diferencias entre regiones para un mismo quintil, pero no permite comparar muy bien la distribución de ingresos en función de la región de destino.

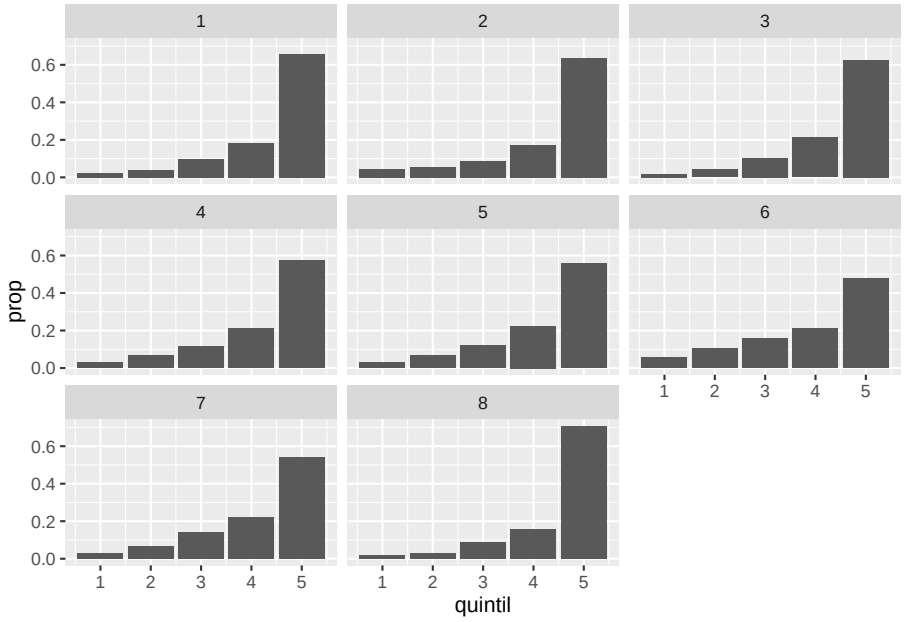
Este problema podría resolverse generando un gráfico por cada región filtrando las observaciones correspondientes.

```
gastos %>%
  filter(region_destino == 1) %>%
  ggplot() +
  geom_bar(aes(quintil, y = stat(prop)),  position = "dodge")
```



Pero sería muchísimo trabajo si tenés que hacer esto para cada una de las 8 regiones. Excepto que ggplot2 tiene una forma de automatizar eso utilizando paneles:

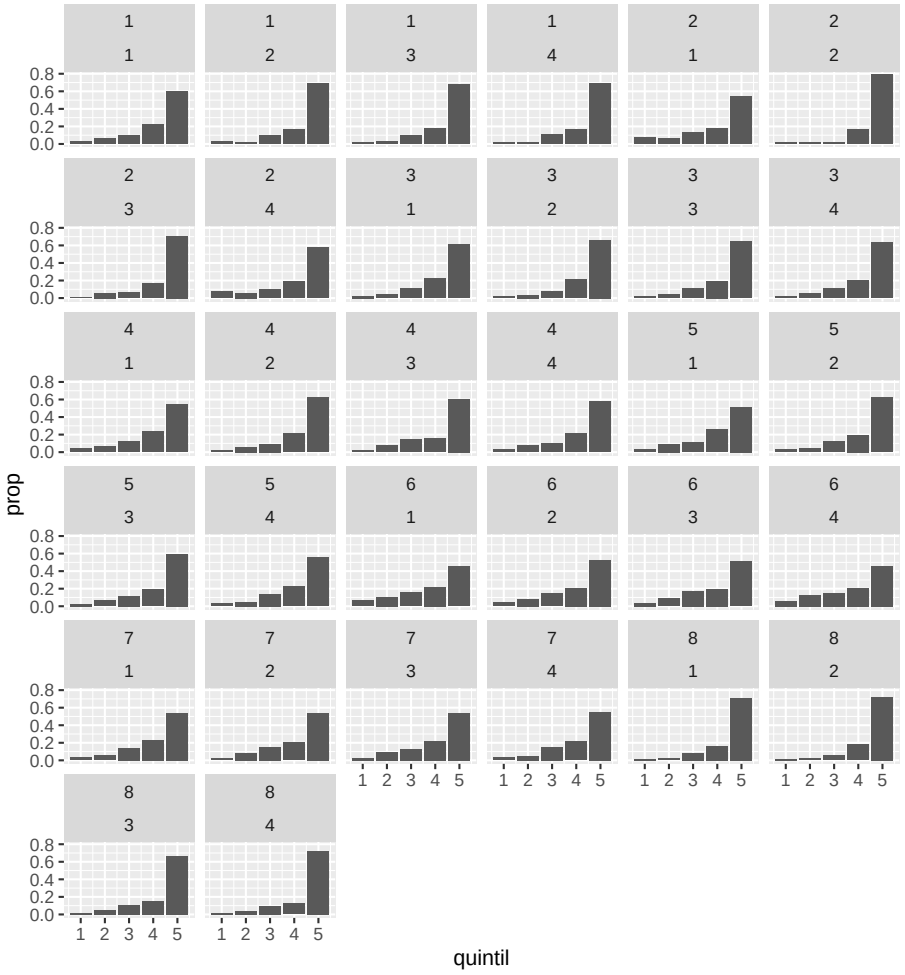
```
ggplot(gastos) +
  geom_bar(aes(quintil, y = stat(prop)),  position = "dodge") +
  facet_wrap(~ region_destino)
```



Esta nueva capa con `facet_wrap()` divide al gráfico inicial en 8 paneles o *facets*, uno por cada región. Esta función requiere saber que variable será la responsable de separar los paneles y para eso se usa la **notación de fórmula** de R: `~ region_destino`. Esto se lee como generar paneles *en función de* `region_destino`.

Una hipótesis razonable podría ser que la distribución de ingresos en cada región también varía según el trimestre. Para ver esto, habría que hacer el gráfico de barras para cada combinación de región de destino y trimestre. En ggplot2 esto se resuelve agregando más variables que definan los paneles "sumando" variables en la fórmula

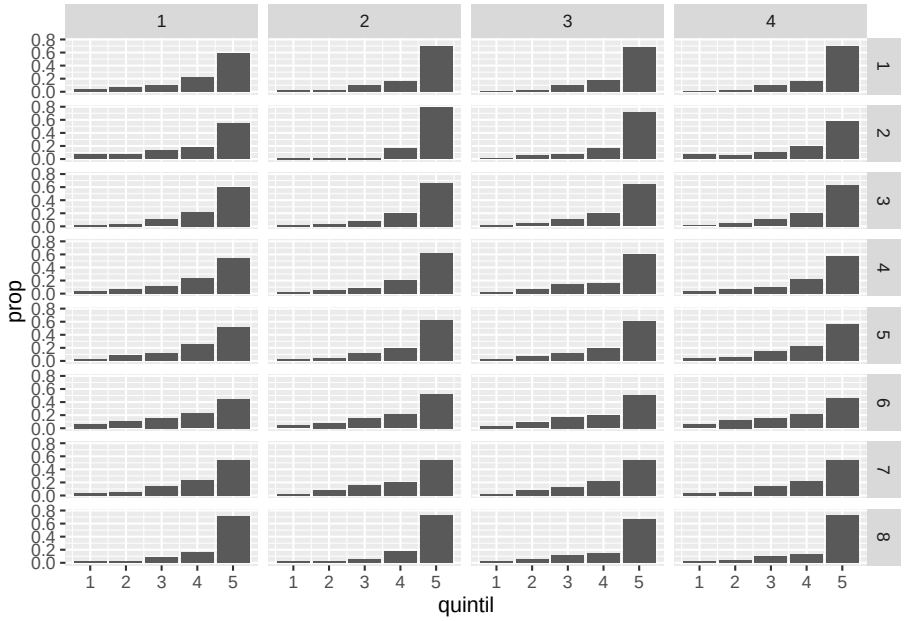
```
ggplot(gastos) +
  geom_bar(aes(quintil, y = stat(prop)),  position = "dodge") +
  facet_wrap(~ region_destino + trimestre)
```



Esto se lee como generar paneles *"en función de* `region_destino` y `trimestre`".

Una alternativa que funciona mejor cuando se hacen paneles en función de dos o más variables es que en vez de organizar los paneles uno luego del otro, tengan una organización. Por ejemplo, que los paneles se organicen en filas según la región de destino y en columna según el trimestre. Para eso hay que reemplazar `facet_wrap()` por `facet_grid()` y cambiar la formula un poco.

```
ggplot(gastos) +
  geom_bar(aes(quintil, y = stat(prop)),  position = "dodge") +
  facet_grid(region_destino ~ trimestre)
```

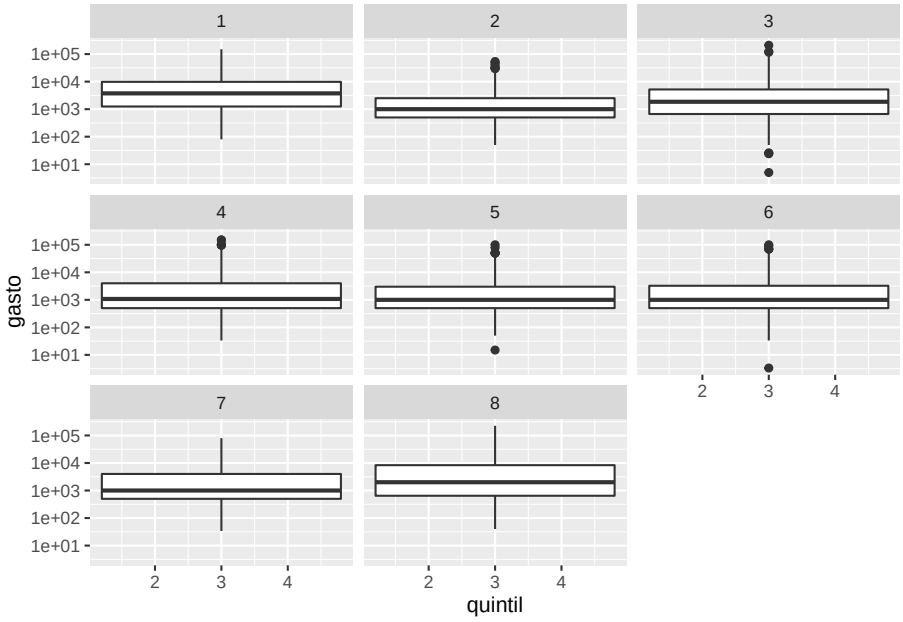


¿Ves como quedan los paneles más organizados y fácil es de leer? Esta organización permite comparar regiones para un trimestre en particular comparando gráficos en la vertical, y comparar trimestres para una misma región leyendo los gráficos en horizontal.

La formula `region_destino ~ trimestre` indica que `region_destino` define las filas y `trimestre` define las columnas.

Tercer desafío

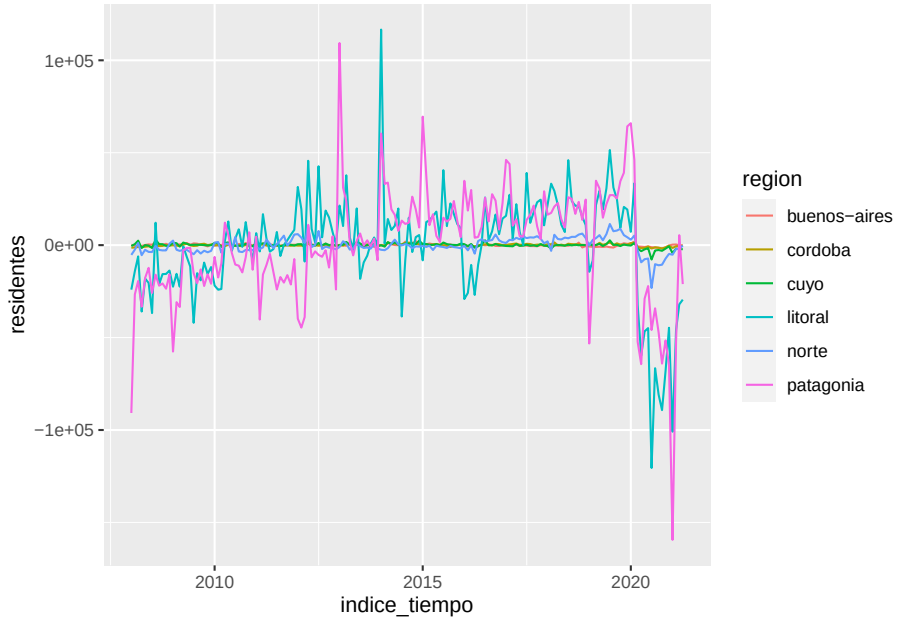
Generá boxplots para analizar como se comporta el `gasto` de cada familia en función del quintil al que pertenecen para cada región.



6.10. Gráficos de líneas suavizadas

Antes viste este gráfico que parece mostrar que la cantidad de visitantes residentes estuvo aumentando entre 2008 y 2020 hasta que las restricciones por la pandemia hicieron que esta cantidad se desplomara.

```
parques %>%
  group_by(trimestre = lubridate::month(indice_tiempo), region) %>%
  mutate(residentes = residentes - mean(residentes)) %>%
  ggplot(aes(indice_tiempo, residentes)) +
  geom_line(aes(color = region))
```



Una forma de guiar al ojo a ver esta tendencia e ignorar las fluctuaciones trimestre a trimestre es usando una línea suave. Las líneas de suavizado ajustan un modelo a los datos y luego grafican las predicciones del modelo. Sin entrar en muchos detalles, se puede aplicar distintos modelos y la elección del mismo dependerá de los datos.

En ggplot2 se puede agregar una línea suave agregando una capa con `geom_smooth()`.

```
parques %>%
  filter(region == "patagonia") %>%
  group_by(trimestre = lubridate::month(indice_tiempo), region) %>%
  mutate(residentes = residentes - mean(residentes)) %>%
  ggplot(aes(indice_tiempo, residentes)) +
  geom_line(aes(color = region)) +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

(Este gráfico además filtra sólo la región de Patagonia para que se vea más claramente.)

Como dice en el mensaje, por defecto `geom_smooth()` suaviza los datos usando el método *loess* (regresión lineal local). Seguramente va a ser muy común que quieras ajustar una regresión lineal global. En ese caso, hay que poner `method = "lm"`:

```
parques %>%
  filter(region == "patagonia") %>%
  group_by(trimestre = lubridate::month(indice_tiempo), region) %>%
  mutate(residentes = residentes - mean(residentes)) %>%
  ggplot(aes(indice_tiempo, residentes)) +
  geom_line(aes(color = region)) +
  geom_smooth(method = "lm")
```

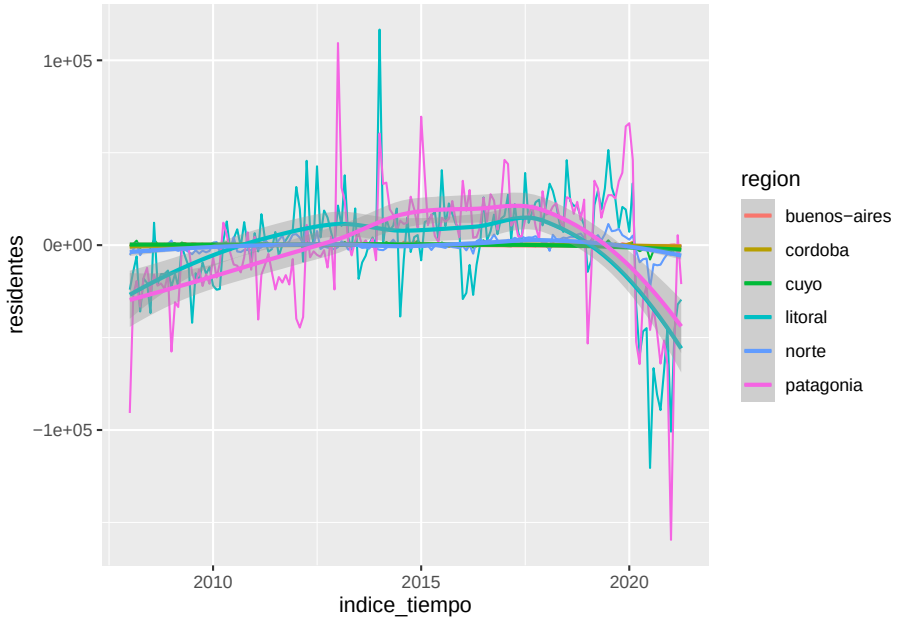
```
## `geom_smooth()` using formula 'y ~ x'
```

El área gris muestra el intervalo de confianza al rededor de este suavizado.

Cómo cualquier geom, podemos modificar el color, el grosor de la línea y casi cualquier cosa que se te ocurra.

```
parques %>%
  group_by(trimestre = lubridate::month(indice_tiempo), region) %>%
  mutate(residentes = residentes - mean(residentes)) %>%
  ggplot(aes(indice_tiempo, residentes)) +
  geom_line(aes(color = region)) +
  geom_smooth(aes(color = region))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```


1e+05
0e+00
−1e+05
residentes
2010
2015
2020
indice_tiempo
region
buenos−aires
cordoba
cuyo
litoral
norte
patagonia

Capítulo 7

Manipulación de datos ordenados usando dplyr y tidyr II

A esta altura del libro vimos varios conjuntos de datos. Cada uno tiene una pinta un poco distinta y trabajar con ellos presenta distintos desafíos. En este capítulo veremos como manipular los datos para organizarlos de distinta manera y que nos sirvan para resolver distintos problemas.

Pero primero algunas definiciones.

Cuando hablamos de datos "ordenados" , o "tidy", o en formato "largo", nos referimos a aquellos set de datos en los cuales:

- cada fila es una observación
- cada columna es una variable

Los datos en formato "ancho" pueden ser muy variados, por lo que son un poco más complejos de definir. Pero la idea general es que:

- cada fila es un "item"
- cada columna es una variable
- muchas veces los nombres de las columnas son variables de los datos

Ancho

pais	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

Largo

pais	anio	casos
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

Una tabla en formato largo va a tener una cierta cantidad de columnas que cumplen el rol de *identificadores* y cuya combinación identifican una única observación y una única columna con el valor de la observación. En el ejemplo de arriba, `pais` y `anio` son las columnas identificadoras y `casos` es la columna que contiene el valor de las observaciones.

En una tabla ancha, cada observación única se identifica a partir de la intersección de filas y columnas. En el ejemplo, los países están en las filas y los años en las columnas.

En general, el formato ancho es más compacto y legible por humanos mientras que el largo es más fácil de manejar con la computadora. Si revisás las tablas de arriba, es más fácil comparar los valores entre países y entre años en la tabla ancha. Pero el nombre de las columnas ("1999", "2000") en realidad ¡son datos! Además este formato se empieza a complicar en cuanto hay más de dos identificadores, como veremos más adelante.

Un mismo conjunto de datos puede ser representado de forma completamente "larga", completamente "ancha" o –lo que es más común– en un formato intermedio. No existe una forma "correcta" de organizar los datos; cada una tiene sus ventajas y desventajas. Por esto es que es muy normal que durante un análisis los datos vayan y vuelvan entre distintos formatos dependiendo de los métodos estadísticos que se le aplican. Entonces, aprender a transformar datos anchos en largos y viceversa es un habilidad muy útil.

Desafío

En las tablas de ejemplo cada país tiene el un valor observado de "casos" para cada año. ¿Cómo agregarías una nueva variable con información sobre "precios"? Dibujá un esquema en papel y lápiz en formato ancho y uno en formato largo. ¿En qué formato es más "natural" esa extensión?

En esta sección vas a usar el paquete **tidyr** para manipular datos. Si no lo tenés instalado, podés hacerlo con el comando:

```
install.packages("tidyr")
```

Cómo siempre, recordá que esto sólo se hace una vez y es recomendable hacerlo desde la consola para que no quede en un bloque de código por accidente.

Y luego cargá tidyr y dplyr (que usaste en una sección anterior) con:

```
library(tidyr)
library(dplyr)
```

7.1. De ancho a largo con pivot_longer()

En secciones anteriores usaste una versión de los datos asociados a parques nacionales. Ahora vas a leer los datos en su formato original:

```
parques_ancho <- readr::read_csv("http://datos.yvera.gob.ar/dataset/458bcbe1-855c-4bc3-a1c9-cd4e8

parques_ancho
```

¿Notaste que en el código anterior no usaste `library(readr)` para cargar el paquete y luego leer? Con la notación `paquete::funcion()` podés acceder a las funciones de un paquete sin tener que cargarlo. Es una buena forma de no tener que cargar un montón de paquetes innecesarios si vas a correr una única función de un paquete pocas veces.

Esta tabla es bastante ancha y puede ser difícil de manejar. Por ejemplo, es complicado hacer series de tiempo de los visitantes de una región porque la información está distribuida entre varias columnas. Tampoco sería simple hacer cuentas por regiones o por tipo de residente con este formato de tabla.

Para convertirlo en una tabla más larga, se usa `pivot_longer()` ("longer" es "más largo" en inglés):

```
parques_largo <- parques_ancho %>%
  pivot_longer(cols = -c("indice_tiempo", "residentes", "no_residentes", "total"),
               names_to = "region_visitante",
               values_to = "valor")
parques_largo
```

```
## # A tibble: 2,952 x 6
##    indice_tiempo residentes no_residentes  total region_visitante          valor
##    <chr>              <dbl>         <dbl>  <dbl> <chr>                    <dbl>
##  1 2008-01           352303        198407 550710 buenos_aires_residentes    885
##  2 2008-01           352303        198407 550710 buenos_aires_no_residen~     0
##  3 2008-01           352303        198407 550710 buenos_aires_total         885
##  4 2008-01           352303        198407 550710 cordoba_residentes         717
##  5 2008-01           352303        198407 550710 cordoba_no_residentes      145
##  6 2008-01           352303        198407 550710 cordoba_total              862
##  7 2008-01           352303        198407 550710 cuyo_residentes           4965
```

```
##  8 2008-01           352303        198407 550710 cuyo_no_residentes          179
##  9 2008-01           352303        198407 550710 cuyo_total                 5144
## 10 2008-01           352303        198407 550710 litoral_residentes       111408
## # ... with 2,942 more rows
```

El primer argumento de`pivot_longer()` es la tabla que va a modificar: `parques_ancho`. El segundo argumento se llama `cols` y es un vector con las columnas que tienen los valores a "alargar". Podría ser un vector escrito a mano (algo como `c("buenos_aires_residentes", "buenos_aires_no_residentes"...)`) pero con más de 20 columnas, escribir todo eso sería tedioso y probablemente estaría lleno de errores. Por eso tidyr provee funciones de ayuda para seleccionar columnas en base a patrones. Por ejemplo `starts_with()` que, como su nombre en inglés lo indica, selecciona las columnas que *empiezan con* una determinada cadena de caracteres. en este caso le decimos que queremos todas las columnas *menos* las que están mencionadas. Por eso hay un "-" antes del vector de columnas. Entonces, el vector `-c("indice_tiempo", "residentes", "no_residentes", "total")` le dice a `pivot_longer()` que seleccione todas las columnas excepto `indice_tiempo`, `residentes`, `no_residentes` y `total`.

Estas funciones accesorias para seleccionar muchas funciones se llaman "tidyselect". Si querés leer más detalles de las distintas formas que podés seleccionar variables leé la documentación usando `?tidyselect::language`.

El tercer y cuarto argumento son los nombres de las columnas de "nombre" y de "valor" que va a tener la nueva tabla. Como la nueva columna de identificación tiene los datos de la región y el tipo de visitante, `region_visitante` es un buen nombre. Y la columna de valor va a tener… bueno, el valor.

Tomate un momento para visualizar lo que acaba de pasar. La tabla ancha tenía un montón de columnas con distintos datos. Ahora estos datos están uno arriba de otro en la columna "valor", pero para identificar el nombre de la columna de la cual vinieron, se agrega la columna "region_visitante".

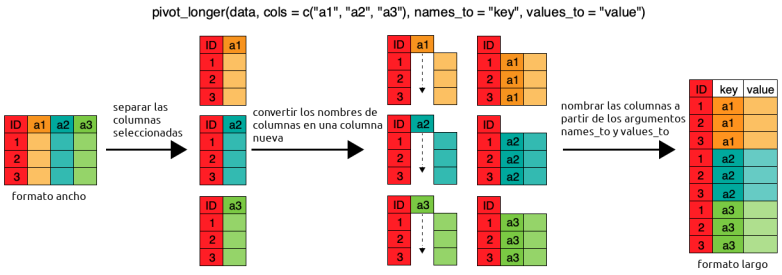


Figura 7.1: Proceso de ancho a largo

La columna `region_visitante` todavía no es muy útil porque contiene 2 datos, la región (Buenos Aires, Cuyo, Litoral, etc.) y el tipo de visitante (Residente,

No Residente o la suma de ambos). Sería mejor separar esta información en dos columnas llamadas "region" y "residente". Para eso está la función `separate()`.

```
parques_largo <- parques_largo %>%
  mutate(region_visitante = stringr::str_replace(region_visitante, "buenos_aires*", "buenos-aires
         region_visitante = stringr::str_replace(region_visitante, "no_residentes", "noresidentes

separate(parques_largo,
         region_visitante,
         into = c("region", "tipo_visitante"),
         sep = "_")
```

```
## # A tibble: 2,952 x 7
##    indice_tiempo residentes no_residentes  total region    tipo_visitante  valor
##    <chr>              <dbl>         <dbl>  <dbl> <chr>     <chr>           <dbl>
##  1 2008-01           352303        198407 550710 buenos-a~ residentes        885
##  2 2008-01           352303        198407 550710 buenos-a~ noresidentes        0
##  3 2008-01           352303        198407 550710 buenos-a~ total             885
##  4 2008-01           352303        198407 550710 cordoba   residentes        717
##  5 2008-01           352303        198407 550710 cordoba   noresidentes      145
##  6 2008-01           352303        198407 550710 cordoba   total             862
##  7 2008-01           352303        198407 550710 cuyo      residentes       4965
##  8 2008-01           352303        198407 550710 cuyo      noresidentes      179
##  9 2008-01           352303        198407 550710 cuyo      total            5144
## 10 2008-01           352303        198407 550710 litoral   residentes     111408
## # ... with 2,942 more rows
```

El primer argumento, como siempre, es la tabla a procesar. El segundo, `col`, es la columna a separar en dos (o más) columnas nuevas. El tercero, `into` es el nombre de las nuevas columnas que `separate()` va a crear. El último argumento es `sep` que define cómo realizar la separación. Por defecto, `sep` es una expresión regular que captura cualquier caracter no alfanumérico. En el caso de `region_visitante` no sirve, porque todos los valores que contienen, por ejemplo "buenos_aires_residentes" usan el "_" para separar palabras. Si hubieamos usado la función `separate()` directamente hubieramos terminado con 3 columnas una con el datos "buenos", otra con el datos "aires" y la última como el dato "residentes". Ni hablar si pensamos en "norte_residentes", nos quedaríamos con 2 columnas. Por esa razón primero tuvimos que manipular las columnas con `mutate()` para reemplazar "buenos_aires" por "buenos-aires" y "no_residentes" por "noresidentes" de manera de poder usar el "_" como separador entre la región y el tipo de visitante únicamente.

Así quedó la columna region_visitante antes de aplicar la separación.

```
parques_largo %>%
  mutate(region_visitante = stringr::str_replace(region_visitante, "buenos_aires*", "bu
         region_visitante = stringr::str_replace(region_visitante, "no_residentes", "no
```

```
## # A tibble: 2,952 x 6
##    indice_tiempo residentes no_residentes  total region_visitante          valor
##    <chr>              <dbl>         <dbl>  <dbl> <chr>                    <dbl>
##  1 2008-01           352303        198407 550710 buenos-aires_residentes     885
##  2 2008-01           352303        198407 550710 buenos-aires_noresident~      0
##  3 2008-01           352303        198407 550710 buenos-aires_total          885
##  4 2008-01           352303        198407 550710 cordoba_residentes          717
##  5 2008-01           352303        198407 550710 cordoba_noresidentes        145
##  6 2008-01           352303        198407 550710 cordoba_total               862
##  7 2008-01           352303        198407 550710 cuyo_residentes            4965
##  8 2008-01           352303        198407 550710 cuyo_noresidentes           179
##  9 2008-01           352303        198407 550710 cuyo_total                 5144
## 10 2008-01           352303        198407 550710 litoral_residentes       111408
## # ... with 2,942 more rows
```

La función `str_replace()` del paquete **stringr** busca un patrón en un texto, por ejemplo "no_residentes" y lo reemplaza por otra cadena de texto que indiquemos, por ejemplo "noresidentes". Este paquete es muy muy útil para manipular texto.

Desafío

Juntá todos los pasos anteriores en una sola cadena de operaciones usando `%>%`.

Nos falta eliminar esas columnas que ya no tienen sentido: residentes, no_resindetes y total. La información de esas columnas están asociadas al formato ancho y podemos volver a generarla si fuera necesario.

Guardemos el resultado de todos los pasos anteriores en `parques_largo`.

```
parques_largo <- parques_ancho %>%
  pivot_longer(cols = -c("indice_tiempo", "residentes", "no_residentes", "total"),
               names_to = "region_visitante",
               values_to = "valor") %>%
  mutate(region_visitante = stringr::str_replace(region_visitante, "buenos_aires*", "bu
         region_visitante = stringr::str_replace(region_visitante, "no_residentes", "no
  separate(region_visitante, into = c("region", "tipo_visitante"), sep = "_") %>%
  select(-c("residentes", "no_residentes", "total"))

parques_largo
```

```
## # A tibble: 2,952 x 4
```

```
##    indice_tiempo region       tipo_visitante  valor
##    <chr>         <chr>        <chr>           <dbl>
##  1 2008-01       buenos-aires residentes         885
##  2 2008-01       buenos-aires noresidentes        0
##  3 2008-01       buenos-aires total              885
##  4 2008-01       cordoba      residentes         717
##  5 2008-01       cordoba      noresidentes       145
##  6 2008-01       cordoba      total              862
##  7 2008-01       cuyo         residentes        4965
##  8 2008-01       cuyo         noresidentes       179
##  9 2008-01       cuyo         total             5144
## 10 2008-01       litoral      residentes      111408
## # ... with 2,942 more rows
```

Si decidimos que este es el formato ideal podríamos guardar los datos en un nuevo archivo csv y luego trabajar directamente con esa versión. Por supuesto, es importante guardar el código que lo genera pero eso puede ir en un archivo separado y se corre una única vez.

7.2. De largo a ancho con `pivot_wider()`

Ahora la variable `parques_largo` está en el formato más largo posible. Tiene 4 columnas, de las cuales sólo una es la columnas con valores. Pero con los datos así no podrías hacer un gráfico de puntos que muestre la relación entre cantidad de visitantes residentes y no residentes en cada mes como en la sección de gráficos.

Muchas veces es conveniente y natural tener los datos en un formato intermedio en donde hay múltiples columnas con los valores de distintas variables observadas.

Pasa "ensanchar" una tabla está la función `pivot_wider()` ("wider" es "más ancha" en inglés) y el código para conseguir este formato intermedio es:

```
parques_medio <- pivot_wider(parques_largo,
                             names_from = tipo_visitante,
                             values_from = valor)
parques_medio
```

```
## # A tibble: 984 x 5
##    indice_tiempo region       residentes noresidentes  total
##    <chr>         <chr>             <dbl>        <dbl>  <dbl>
##  1 2008-01       buenos-aires        885            0    885
##  2 2008-01       cordoba             717          145    862
```

```
##  3 2008-01       cuyo              4965          179   5144
##  4 2008-01       litoral         111408        55335 166743
##  5 2008-01       norte             4241          774   5016
##  6 2008-01       patagonia       230087       141973 372060
##  7 2008-02       buenos-aires       624            0    624
##  8 2008-02       cordoba            475          148    623
##  9 2008-02       cuyo              4803          139   4942
## 10 2008-02       litoral          85853        53596 139449
## # ... with 974 more rows
```

Nuevamente el primer argumento es la tabla original. El segundo, `names_from` es la columna cuyos valores únicos van a convertirse en nuevas columnas. La columna `tipo_visitante` tiene los valores `"residentes"`, `"noresidentes"` y `"total"` y entonces la tabla nueva tendrá tres columnas con esos nombres. El tercer argumento, `values_from`, es la columna de la cual sacar los valores.

Para volver al formato más ancho, basta con agregar más columnas en el argumento `names_from`:

```
pivot_wider(parques_largo,
            names_from = c(region, tipo_visitante),
            names_sep = "_",
            values_from = valor)
```

```
## # A tibble: 164 x 19
##    indice_tiempo `buenos-aires_residentes` `buenos-aires_nore~ `buenos-aires_to~
##    <chr>                             <dbl>                <dbl>             <dbl>
##  1 2008-01                             885                    0               885
##  2 2008-02                             624                    0               624
##  3 2008-03                               0                    0                 0
##  4 2008-04                             462                    0               462
##  5 2008-05                            1091                    0              1091
##  6 2008-06                            1126                    0              1126
##  7 2008-07                            1304                    0              1304
##  8 2008-08                            2912                    0              2912
##  9 2008-09                            1394                    0              1394
## 10 2008-10                            2004                    0              2004
## # ... with 154 more rows, and 15 more variables: cordoba_residentes <dbl>,
## #   cordoba_noresidentes <dbl>, cordoba_total <dbl>, cuyo_residentes <dbl>,
## #   cuyo_noresidentes <dbl>, cuyo_total <dbl>, litoral_residentes <dbl>,
## #   litoral_noresidentes <dbl>, litoral_total <dbl>, norte_residentes <dbl>,
## #   norte_noresidentes <dbl>, norte_total <dbl>, patagonia_residentes <dbl>,
## #   patagonia_noresidentes <dbl>, patagonia_total <dbl>
```

En esta llamada también está el argumento `names_sep`, que determina el caracter que se usa para crear el nombre de las nuevas columnas, usamos `"_"` para que quede igual al original.

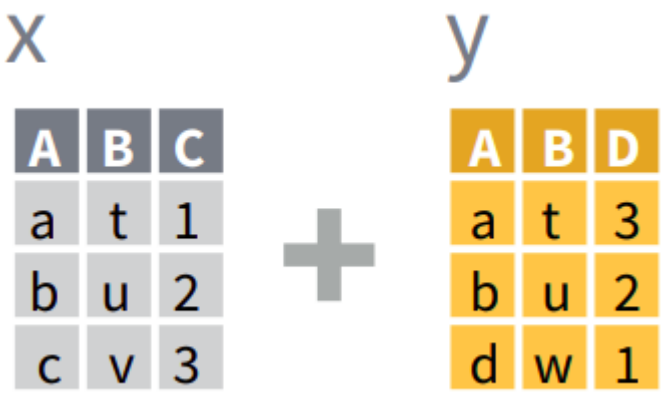
Desafío

- ¿Cómo es la tabla más ancha posible que podés generar con estos datos? ¿Cuántas filas y columnas tiene? No es necesario que lo intentes hacer ahora pero siempre sirve hacer un diagrama para organizar las ideas.

7.3. Uniendo tablas

Hasta ahora todo lo que usaste de dplyr involucra trabajar y modificar con una sola tabla a la vez, pero es muy común tener dos o más tablas con datos relacionados. En ese caso, tenemos que *unir* estas tablas a partir de una o más variables en común o *keys*. En Excel u otro programa de hojas de cálculo, esto se resuelve con la función "VLOOKUP" o "BUSCARV", en R y en particular dentro del mundo de dplyr hay que usar la familia de funciones `*_join()`. Hay una función para cada tipo de unión que queramos hacer.

Asumiendo que querés unir dos data.frames o tablas `x` e `y` que tienen en común una variable `A`:



- `full_join()`: devuelve todas las filas y todas las columnas de ambas tablas `x` e `y`. Cuando no coinciden los elementos en `y`, devuelve `NA` (dato faltante). Esto significa que no se pierden filas de ninguna de las dos tablas aún cuando no hay coincidencia. Está es la manera más segura de unir tablas.

- `left_join()`: devuelve todas las filas de `x` y todas las columnas de `x` e `y`. Las filas en `x` que no tengan coincidencia con `y` tendrán `NA` en las nuevas columnas. Si hay múltiples coincidencias entre `x`e `y`, devuelve todas las coincidencias posibles.

- `right_join()`: es igual que `left_join()` pero intercambiando el orden de `x` e `y`. En otras palabras, `right_join(x, y)` es idéntico a `left_join(y, x)`.
- `inner_join()`: devuelve todas las filas de `x` donde hay coincidencias con `y` y todas las columnas de `x` e `y`. Si hay múltiples coincidencias entre `x` e `y`, entonces devuelve todas las coincidencias. Esto significa que eliminará las filas (observaciones) que no coincidan en ambas tablas, lo que puede ser peligroso.

Ahora vamos a seguir trabajando con las base de datos de `parques_largo` y de paso unirlo a una nueva base de datos `hoteles` que contiene información de viajeros por región de destino y origen a lo largo del tiempo.

Para que los datos sean más manejables y poder ver que es lo que sucede vamos a quedarnos sólo con la información de 2018.

```
library(lubridate)
library(stringr)

parques_2018 <- parques_largo %>%
  mutate(indice_tiempo = ym(indice_tiempo)) %>%
  filter(year(indice_tiempo) == 2018)

hoteles <- readr::read_csv("http://datos.yvera.gob.ar/dataset/93db331e-6970-4d74-8589-

hoteles_2018 <- hoteles %>%
  mutate(region_de_destino = tolower(region_de_destino) %>%
           str_replace(" ", "-") %>%
           str_replace("ó", "o")) %>%
  pivot_wider(names_from  = origen_viajeros, values_from = viajeros) %>%
  rename(hoteles_noresidentes = `No residentes`,
         hoteles_residentes = Residentes,
         region = region_de_destino) %>%
  mutate(indice_tiempo = lubridate::ym(indice_tiempo)) %>%
  filter(lubridate::year(indice_tiempo) == 2018)
```

En el código de más arriba volvimos a usar la librería lubridate. En este caso la función `year()` permite extraer el parte del año de la fecha que está guardada en la columna `indice_tiempo`. Esto luego nos pemite filtrar los datos por años.

Esta nueva tabla tiene 5 columnas: `indice_tiempo` con la fecha (año y mes), `region` con la regiones de destino de los visitantes, `observaciones` con muchos NA, `hoteles_residentes` y `hoteles_noresidentes` que indica la cantidad de personas resindetes y no residentes que se alojaron en hoteles. Las columnas `region` e `indice_tiempo` también están presente en la tabla `parques_2018` y son las que van a servir como variables llave para unir las dos tablas.

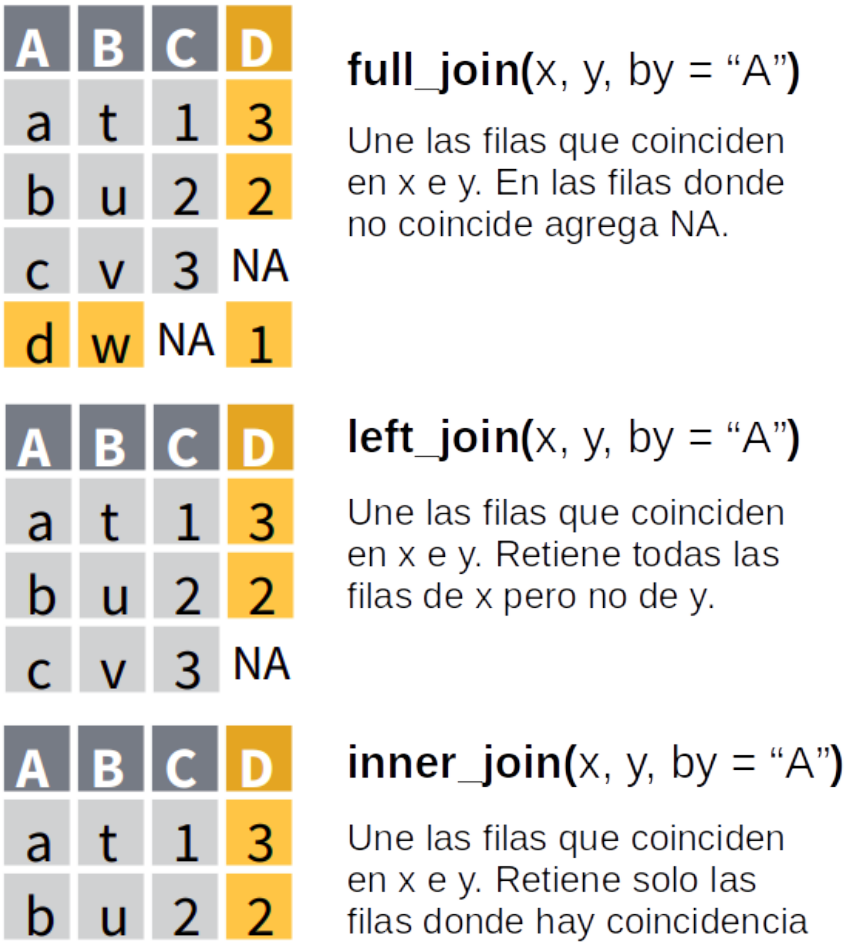


Figura 7.2: Familia de uniones de dplyr

Para unir las dos tablas, cualquier función *join* requiere cierta información:

- las tablas a unir: son los dos primeros argumentos.
- qué variable o variables (se puede usar más de una!) usar para identificar coincidencias: el argumento `by`.

Unamos `parques_2018` y `hoteles_2018` primero con `full_join()`:

```
parques_hoteles_2018 <- full_join(parques_2018, hoteles_2018,
                                  by = c("region", "indice_tiempo"))
parques_hoteles_2018
```

```
## # A tibble: 228 x 7
##    indice_tiempo region       tipo_visitante  valor observaciones hoteles_residen~
##    <date>        <chr>        <chr>           <dbl> <chr>                    <dbl>
##  1 2018-01-01    buenos-aires residentes        239 <NA>                    385396
##  2 2018-01-01    buenos-aires noresidentes        4 <NA>                    385396
##  3 2018-01-01    buenos-aires total              243 <NA>                    385396
##  4 2018-01-01    cordoba      residentes        2663 <NA>                    299440
##  5 2018-01-01    cordoba      noresidentes       239 <NA>                    299440
##  6 2018-01-01    cordoba      total             2902 <NA>                    299440
##  7 2018-01-01    cuyo         residentes        4959 <NA>                    141927
##  8 2018-01-01    cuyo         noresidentes       259 <NA>                    141927
##  9 2018-01-01    cuyo         total             5218 <NA>                    141927
## 10 2018-01-01    litoral      residentes      157509 <NA>                    261555
## # ... with 218 more rows, and 1 more variable: hoteles_noresidentes <dbl>
```

Si miramos de cerca la tabla unida veremos un par de cosas:

- Todas las columnas de `parques_2018` y de `hoteles2018` están presentes.
- Todas las observaciones están presentes, aún las regiones que están presentes en `hoteles_2018` pero no en `parques_2018` (no hay parques nacionales en CABA). En esos casos ahora tenemos `NA` en las columnas que vienen del data.frame de parques. Esto genera una tabla con 228 filas.

Esta es la opción más segura si no sabemos si todas las observaciones de una tabla están presente en a otra.

Si solo nos interesa conservar las filas de la tabla *de la izquierda*, en este caso `parques_2018` entonces:

```
parques_hoteles_2018 <- left_join(parques_2018, hoteles_2018,
                                  by = c("region", "indice_tiempo"))
parques_hoteles_2018
```

```
## # A tibble: 216 x 7
##    indice_tiempo region       tipo_visitante  valor observaciones hoteles_residen~
##    <date>        <chr>        <chr>           <dbl> <chr>                     <dbl>
##  1 2018-01-01    buenos-aires residentes        239 <NA>                     385396
##  2 2018-01-01    buenos-aires noresidentes        4 <NA>                     385396
##  3 2018-01-01    buenos-aires total              243 <NA>                     385396
##  4 2018-01-01    cordoba      residentes        2663 <NA>                     299440
##  5 2018-01-01    cordoba      noresidentes       239 <NA>                     299440
##  6 2018-01-01    cordoba      total             2902 <NA>                     299440
##  7 2018-01-01    cuyo         residentes        4959 <NA>                     141927
##  8 2018-01-01    cuyo         noresidentes       259 <NA>                     141927
##  9 2018-01-01    cuyo         total             5218 <NA>                     141927
## 10 2018-01-01    litoral      residentes      157509 <NA>                     261555
## # ... with 206 more rows, and 1 more variable: hoteles_noresidentes <dbl>
```

Ahora esperamos que la tabla resultante tenga la misma cantidad de filas que `parques_2018` y efectivamente eso ocurre. Todas las regiones en `parques_2018` tienen coincidencia con `hoteles_2018` y por eso no hay NAs en las columnas que vienen de ese último data.frame.

Finalmente, si quisiéramos quedarnos solo con las observaciones que están presentes en ambas tablas usamos `inner_join()`.

```
parques_hoteles_2018 <- inner_join(parques_2018, hoteles_2018,
                                   by = c("region", "indice_tiempo"))
parques_hoteles_2018
```

```
## # A tibble: 216 x 7
##    indice_tiempo region       tipo_visitante  valor observaciones hoteles_residen~
##    <date>        <chr>        <chr>           <dbl> <chr>                     <dbl>
##  1 2018-01-01    buenos-aires residentes        239 <NA>                     385396
##  2 2018-01-01    buenos-aires noresidentes        4 <NA>                     385396
##  3 2018-01-01    buenos-aires total              243 <NA>                     385396
##  4 2018-01-01    cordoba      residentes        2663 <NA>                     299440
##  5 2018-01-01    cordoba      noresidentes       239 <NA>                     299440
##  6 2018-01-01    cordoba      total             2902 <NA>                     299440
##  7 2018-01-01    cuyo         residentes        4959 <NA>                     141927
##  8 2018-01-01    cuyo         noresidentes       259 <NA>                     141927
##  9 2018-01-01    cuyo         total             5218 <NA>                     141927
## 10 2018-01-01    litoral      residentes      157509 <NA>                     261555
## # ... with 206 more rows, and 1 more variable: hoteles_noresidentes <dbl>
```

En este caso, perdemos las filas de `hoteles_2018` que no encontraron coincidencia en `parques_2018`, es decir la región de CABA. La tabla resultante es igual a la tabla que generamos en el ejemplo anterior con 216 filas.

Desafío

Ahora es tu turno. Nos faltó revisar la función `right_join()`. Intentá unir las dos tablas con las que estuvimos trabajando cambiando el orden en el que las llamas en la función.

1. ¿Ves alguna diferencia en los resultados?
2. Si encontras alguna diferencia, intentá explicar a que se debe. Si no hay diferencias, pensá por qué.

Hasta ahora en las uniones usamos columnas que tenian el mismo nombre en ambas tablas. ¿Qué ocurre cuando las columnas no se llaman igual?. Una solución es renombrar las columnas en alguna de la tablas (como en los ejemplos). La otra opción es indicar como se llaman las columnas que tenemos que unir en el parámetro `by` de la función join. Por ejemplo, si en la tabla parques region se llama *region* y en la tabla hoteles se llama *regiones*, el parámetro by debe ser especificado como: `by = c("region" = "regiones")`

Capítulo 8

Tablas

Por defecto, la forma en que R Markdown muestra tablas es bastante feo porque muestra lo que verías si corrieras el código en la consola.

```
library(dplyr)
library(tidyr)
library(lubridate)

parques_ancho <- readr::read_csv("http://datos.yvera.gob.ar/dataset/458bcbe1-855c-4bc3-a1c9-cd4e8

parques_largo <- parques_ancho %>%
  pivot_longer(cols = -c("indice_tiempo", "residentes", "no_residentes", "total"),
               names_to = "region_visitante",
               values_to = "valor") %>%
  mutate(region_visitante = stringr::str_replace(region_visitante, "buenos_aires*", "buenos-aires
         region_visitante = stringr::str_replace(region_visitante, "no_residentes", "noresidentes
  separate(region_visitante, into = c("region", "tipo_visitante"), sep = "_") %>%
  select(-c("residentes", "no_residentes", "total"))

parques_resumen <-  parques_largo %>%
  mutate(indice_tiempo = ym(indice_tiempo)) %>%
  group_by(anio = year(indice_tiempo), region, tipo_visitante) %>%
  summarise(valor = mean(valor)) %>%
  filter(region %in% c("cordoba", "cuyo", "patagonia"),
         anio >= 2020) %>%
  pivot_wider(names_from = "tipo_visitante", values_from = "valor")
```

```
parques_resumen
```

```
## # A tibble: 6 x 5
```

```
## # Groups:   anio, region [6]
##    anio region    noresidentes residentes  total
##   <dbl> <chr>            <dbl>      <dbl>  <dbl>
## 1  2020 cordoba          36.9        626.   663.
## 2  2020 cuyo             55.2       1104.  1160.
## 3  2020 patagonia     25615.       66082  91697.
## 4  2021 cordoba           8.38       958.   966.
## 5  2021 cuyo              3.62      1586.  1590.
## 6  2021 patagonia       113.       77502. 77615
```

8.1. Tablas simples con `kable`

Existen varios paquetes para mostrar tablas lindas pero una forma simple y sin vueltas es usando la función `kable()` del paquete **knitr**. Sólo usando esta función ya devuelve una tabla con un diseño más limpio y acabado:

```
library(knitr)
kable(parques_resumen)
```

anio	region	noresidentes	residentes	total
2020	cordoba	36.91667	626.1667	663.0833
2020	cuyo	55.25000	1104.4167	1159.6667
2020	patagonia	25615.25000	66082.0000	91697.1667
2021	cordoba	8.37500	958.1250	966.5000
2021	cuyo	3.62500	1586.1250	1589.7500
2021	patagonia	112.62500	77502.3750	77615.0000

La mayoría de las veces, el nombre de las columnas que queremos mostrar no va a ser igual que el nombre de las columnas en R. En la tabla `parques_resumen`, los nombres están en minúscula, sin espacios y sin "caracteres especiales" como "ñ". Esto es útil para comunicarse con R, pero no está bueno para comunicarse con otras personas. El argumento `col.names` permite especificar el nombre de las columnas para mostrar:

```
kable(parques_resumen,
      col.names = c("Año", "Región", "No Residentes", "Residentes", "Total"))
```

Año	Región	No Residentes	Residentes	Total
2020	cordoba	36.91667	626.1667	663.0833
2020	cuyo	55.25000	1104.4167	1159.6667
2020	patagonia	25615.25000	66082.0000	91697.1667
2021	cordoba	8.37500	958.1250	966.5000
2021	cuyo	3.62500	1586.1250	1589.7500
2021	patagonia	112.62500	77502.3750	77615.0000

En Español, en general usamos la coma para separar los decimales y el punto como separador de miles. Además, esta tabla tiene el problema de la precisión innecesaria: ¿realmente tiene sentido reportar el promedio de visitantes con 5 decimales? Para eso, hay que modificar el argumento `format.args` de esta manera:

```
kable(parques_resumen,
      col.names = c("Año", "Región", "No Residentes", "Residentes", "Total"),
      format.args = list(decimal.mark = ",", big.mark = ".", digits = 1, scientific = FALSE))
```

Año	Región	No Residentes	Residentes	Total
2.020	cordoba	37	626	663
2.020	cuyo	55	1.104	1.160
2.020	patagonia	25.615	66.082	91.697
2.021	cordoba	8	958	966
2.021	cuyo	4	1.586	1.590
2.021	patagonia	113	77.502	77.615

En esta llamada hay varios elementos que son parte del argumento `format.args`:

- `decimal.mark = ","`: usar la coma para la marca de decimales,
- `big.mark = "."`: usar el punto para separador de miles,
- `digits = 1`: redondear todo a 1 cifra significativa,
- `scientific = FALSE`: no usar notación científica.

Uff... pero ponerle el punto de miles a los años queda raro. La solución es convertir la columna `anio` en caracter. De esta manera, no se ve afectada por el formato numérico.

```
parques_resumen %>%
  mutate(anio = as.character(anio)) %>%
  kable(col.names = c("Año", "Región", "No Residentes", "Residentes", "Total"),
        format.args = list(decimal.mark = ",", big.mark = ".", digits = 1, scientific = FALSE))
```

Año	Región	No Residentes	Residentes	Total
2020	cordoba	37	626	663
2020	cuyo	55	1.104	1.160
2020	patagonia	25.615	66.082	91.697
2021	cordoba	8	958	966
2021	cuyo	4	1.586	1.590
2021	patagonia	113	77.502	77.615

Un detalle final podría ser convertir en mayúsculas los nombres de las regiones. Para eso se puede usar la función `str_to_sentence()` del paquete **stringr**.

```
parques_resumen %>%
  mutate(anio = as.character(anio),
         region = stringr::str_to_sentence(region)) %>%
  kable(col.names = c("Año", "Región", "No Residentes", "Residentes", "Total"),
        format.args = list(decimal.mark = ",", big.mark = ".", digits = 1, scientific =
```

Año	Región	No Residentes	Residentes	Total
2020	Cordoba	37	626	663
2020	Cuyo	55	1.104	1.160
2020	Patagonia	25.615	66.082	91.697
2021	Cordoba	8	958	966
2021	Cuyo	4	1.586	1.590
2021	Patagonia	113	77.502	77.615

`kable` es una función diseñada para ser simple y hacer pocas cosas, para hacer cosas un poco mas complicadas está el paquete **kableExtra**.

8.2. Supertablas con kableExtra

El paquete kableExtra, como su nombre lo indica, nació para extender el poder de la función `kable()`.

```
library(kableExtra)  # instalar con install.packages("kableExtra")
```

Primero, guardamos la tabla anterior para editarla con las operaciones de kableExtra.

```
parques_tabla <- parques_resumen %>%
  mutate(anio = as.character(anio),
         region = stringr::str_to_sentence(region)) %>%
  kable(col.names = c("Año", "Región", "No Residentes", "Residentes", "Total"),
        format.args = list(decimal.mark = ",", big.mark = ".", digits = 1, scientific =
```

La primera columna de la tabla de arriba es un tanto redundante. Sería mejor agrupar las filas según el año. Esto se consigue con la función `collapse_rows()`:

```
parques_tabla %>%
  collapse_rows(columns = 1, valign = "top")  %>%
  kable_styling()
```

El primer argumento es el número de la columna que queremos agrupar. En este caso, la columna `anio` es la primera. El segundo argumento es la alineación

Año	Región	No Residentes	Residentes	Total
2020	Cordoba	37	626	663
	Cuyo	55	1.104	1.160
	Patagonia	25.615	66.082	91.697
2021	Cordoba	8	958	966
	Cuyo	4	1.586	1.590
	Patagonia	113	77.502	77.615

Año	Región	No Residentes	Residentes	Total
2020	Cordoba	37	626	663
	Cuyo	55	1.104	1.160
	Patagonia	25.615	66.082	91.697
2021	Cordoba	8	958	966
	Cuyo	4	1.586	1.590
	Patagonia	113	77.502	77.615

vertical. Por defecto, `collapse_rows()` pone las etiquetas en el centro, pero la convención más general es ponerlas arriba.

¿Qué es esa `kable_styling()`? kableExtra permite cambiar el estilo de las tablas muy fácilmente. `kable_styling()` es el estilo por defecto, que es igual al que produce `kable()`, pero existen muchos otros. Si viste tablas hechas en LaTeX, quizás este estilo te resulte familiar:

```
parques_tabla %>%
  collapse_rows(columns = 1, valign = "top")  %>%
  kable_classic_2()
```

La función `column_spec()` permite cambiar los parámetros gráficos de una o más columnas. En este caso cambiamos la segunda columna (`column = 2`) para que las palabras aparezcan italizadas (`italic = TRUE`):

```
parques_tabla %>%
  column_spec(column = 2, italic = TRUE) %>%
  collapse_rows(columns = 1, valign = "top")  %>%
  kable_styling()
```

Lo notable de `column_spec()` es que en sus argumentos se pueden poner vectores de manera de generar formato condicional. Para resaltar con negrita el año 2021:

```
parques_tabla %>%
  column_spec(column = 1, bold = (parques_resumen$anio == 2021)) %>%
  collapse_rows(columns = 1, valign = "top")  %>%
  kable_styling()
```

Año	Región	No Residentes	Residentes	Total
2020	*Cordoba*	37	626	663
	Cuyo	55	1.104	1.160
	Patagonia	25.615	66.082	91.697
2021	*Cordoba*	8	958	966
	Cuyo	4	1.586	1.590
	Patagonia	113	77.502	77.615

Año	Región	No Residentes	Residentes	Total
2020	Cordoba	37	626	663
	Cuyo	55	1.104	1.160
	Patagonia	25.615	66.082	91.697
2021	Cordoba	8	958	966
	Cuyo	4	1.586	1.590
	Patagonia	113	77.502	77.615

Conviene detenerse para mirar un poco con detalle el código. La llamada a `column_spec()` está cambiando el formato de la columna 1. En particular, va a determinar si el texto está en negrita (bold) o no. Y la forma para determinarlo es el vector `parques_resumen$anio == 2021`. ¿Qué es ese vector?

```
parques_resumen$anio == 2021
```

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

Por lo tanto, `column_spec(column = 1, bold = (parques_resumen$anio == 2021))` va a hacer que las tres últimas filas de la columna uno estén en negrita, y las demás no. Luego, `collapse_rows()` colapsa las filas según el año y entonces esas tres últimas filas se transforman en una.

En este caso el órden de las operaciones es importante. Si primero colapsamos y luego damos formato, el resultado no es el esperado:

```
parques_tabla %>%
  collapse_rows(columns = 1, valign = "top")  %>%
  column_spec(column = 1, bold = (parques_resumen$anio == 2021)) %>%
  kable_styling()
```

```
## Usually it is recommended to use column_spec before collapse_rows, especially in La
```

Esto da un montón de flexibilidad en el estilo del texto. Por ejemplo, se puede hacer que el color del texto cambie según el valor usando `column_spec()` y `spec_color()`.

Año	Región	No Residentes	Residentes	Total
2020	Cordoba	37	626	663
	Cuyo	55	1.104	1.160
	Patagonia	25.615	66.082	91.697
2021	Cordoba	8	958	966
	Cuyo	4	1.586	1.590
	Patagonia	113	77.502	77.615

Año	Región	No Residentes	Residentes	Total
2020	Cordoba	37	626	663
	Cuyo	55	1.104	1.160
	Patagonia	25.615	66.082	91.697
2021	Cordoba	8	958	966
	Cuyo	4	1.586	1.590
	Patagonia	113	77.502	77.615

```
parques_tabla %>%
  column_spec(5, color = spec_color(parques_resumen$total)) %>%
  collapse_rows(columns = 1, valign = "top")  %>%
  kable_styling()
```

La función nueva, `spec_color()`, genera colores a partir de un vector. Podés ver cómo funciona en esta linea:

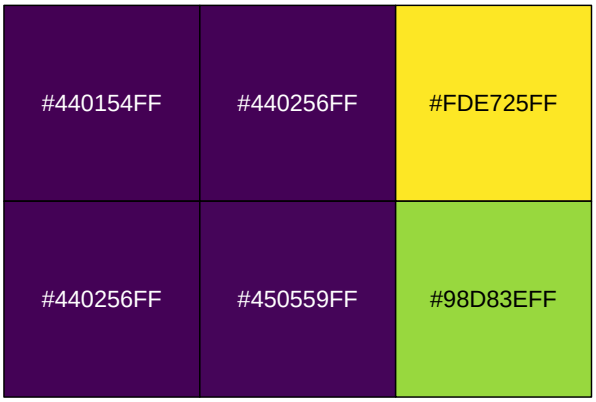
```
spec_color(parques_resumen$total)
```

```
## [1] "#440154FF" "#440256FF" "#FDE725FF" "#440256FF" "#450559FF" "#98D83EFF"
```

Esos números son representaciones en hexadecimal de los colores que ves en la tabla.

```
scales::show_col(spec_color(parques_resumen$total))
```

Año	Región	No Residentes	Residentes	Total
2020	Cordoba	37	626	663
2020	Cuyo	55	1.104	1.160
2020	Patagonia	25.615	66.082	91.697
2021	Cordoba	8	958	966
2021	Cuyo	4	1.586	1.590
2021	Patagonia	113	77.502	77.615



Así como se puede cambiar el estilo de las columnas con `column_spec()`, se puede cambiar el estilo de las filas con `row_spec()`. Por ejemplo, se puede resaltar cada 2 líneas con este código:

```
parques_tabla %>%
  row_spec(row = seq(2, nrow(parques_resumen), by = 2), background = "aquamarine") %>%
  collapse_rows(columns = 1, valign = "top")  %>%
  kable_styling()
```

`row_spec()` está cambiando el fondo (background) al color "aquamarine" (que es ese turquesa) a las filas indicadas por `seq(2, nrow(paises_seleccion), by = 2)`. ¿Qué es eso? `seq()` genera una secuencia (de **seq**uence) de números empezando en 2, terminando en la cantidad de filas de `parques_resumen` y saltando de a dos. En resumen:

Año	Región	No Residentes	Residentes	Total
2020	Cordoba	37	626	663
	Cuyo	55	1.104	1.160
	Patagonia	25.615	66.082	91.697
2021	Cordoba	8	958	966
	Cuyo	4	1.586	1.590
	Patagonia	113	77.502	77.615

```
seq(2, nrow(parques_resumen), by = 2)
```

```
## [1] 2 4 6
```

Como caso especial, se puede usar `row = 0` en `row_spec()` para cambiar el formato del encabezado.

```
parques_tabla %>%
  row_spec(row = 0, underline = TRUE) %>%
  collapse_rows(columns = 1, valign = "top")  %>%
  kable_styling()
```

Capítulo 9

Apariencia de gráficos

En el capítulo de ggplot2 vimos el funcionamiento básico de esta librería. Los gráficos generados en ese capítulo son los típicos gráficos exploratorios: rápidos de hacer, pero sin mucho tiempo dedicado a los detalles. No son gráficos que estén listos para presentar en público.

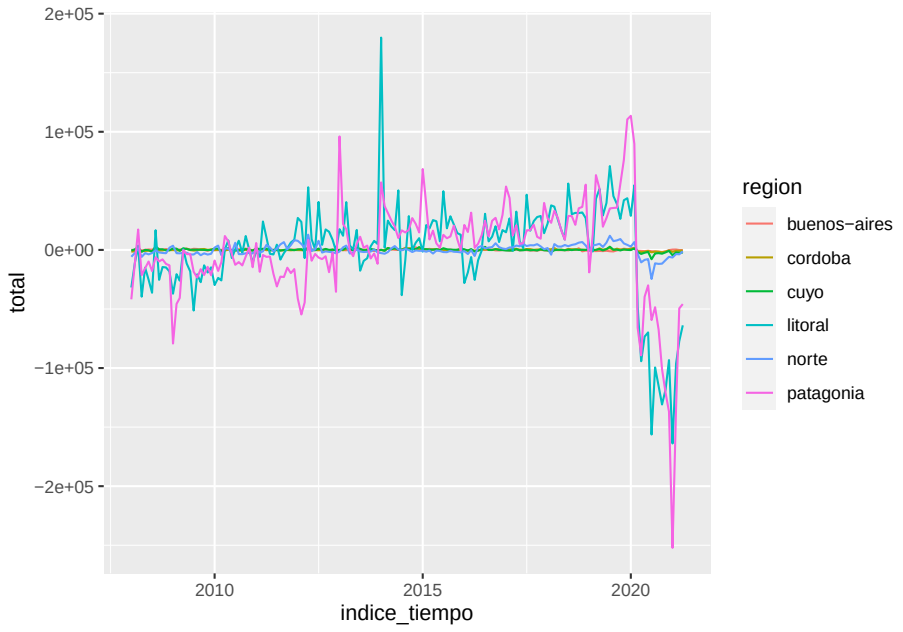
Modificar la apariencia de un gráfico no es sólo cuestión de cambiarle los colores o la fuente para que sea "lindo". Implica pensar sobre cómo hacer que el mensaje del gráfico sea fácil de entender rápidamente.

En este capítulo, vamos a empezar con un gráfico de líneas que hicimos antes que muestra la evolución de la anomalía trimestral de la cantidad total de visitantes a parques nacionales de distintas regiones.

```
library(dplyr)
library(ggplot2)

parques <- readr::read_csv("datos/parques_tidy.csv") %>%
  group_by(mes = lubridate::month(indice_tiempo), region) %>%
  mutate(total = total - mean(total)) %>%
  ungroup()

parques %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line(aes(color = region))
```



Pero este gráfico no es algo que podamos publicar. Los números del eje y en notación científica no son muy amigables, los títulos en los ejes son nombres de variables en minúscula y sin espacios y la escala de colores es medio fea.

Vamos a ir mejorando este gráfico para que sea mucho más presentable y fácil de leer.

Desafío

Antes de seguir analizá el bloque de código anterior. ¿Podés entender todo lo que hace cada línea? Tratá de describirlo en tus propias palabras.

9.1. Escalas

Cuando ggplot2 *mapea* distintos colores a los distintos valores de la columna `region`, lo que define qué color le corresponde a qué categoría de `región` es una **escala**. Esto no se limita a los colores; detrás de todo *mapeo* generado por `aes()` hay una escala. Si un gráfico no tiene una escala explícita, ggplot2 usa las escalas por defecto.

Para modificar una escala, hay que sumar una nueva capa con una función de escala. Así como las funciones de geometrías empiezan con `geom_`, las funciones de escala comienzan con `scale_` (de escala en inglés). Luego, sigue el tipo de apariencia que mapean (`color`, `fill`, `shape`, etc.) y en muchos casos un nombre o una característica de esa escala.

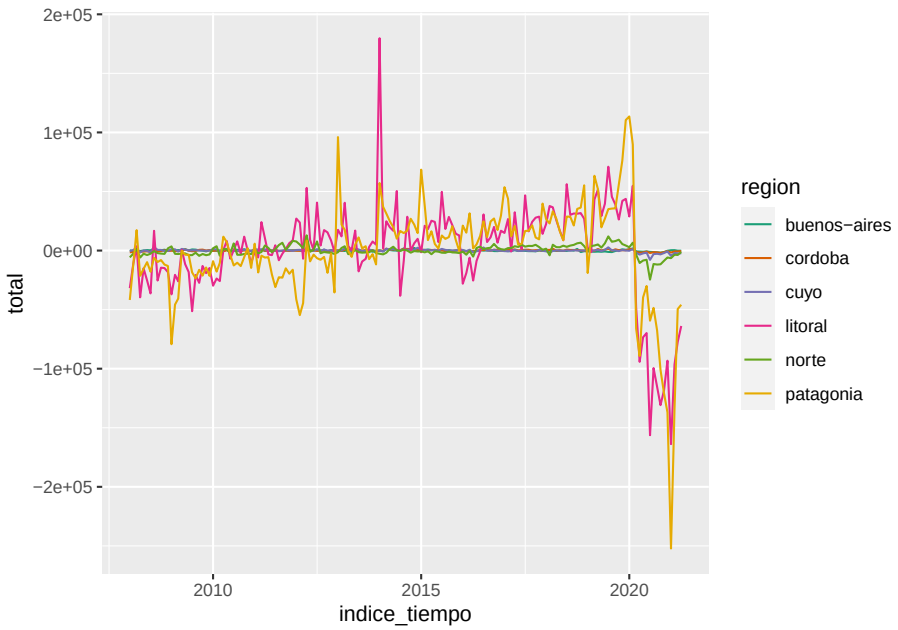
Es importante distinguir entre las escalas continuas y discretas. Las escalas continuas mapean valores continuos como números, o fechas, mientras que las discretas, mapean valores categóricos. En el caso de nuestro gráfico, donde los colores representan distintas regiones del país, se trata de una variable categórica, por lo que necesitamos escalas discretas.

9.1.1. Escalas de colores

La escala de colores que usa ggplot2 por defecto no es de las mejores, de hecho las personas que tienen daltonismo muy posiblemente no logren diferenciar todas las líneas. Una escala o paleta de colores usada (principalmente para valores continuos) es **viridis** que fue creada justamente para resolver este y otros problemas. Otra gran familia de paletas de colores es **ColorBrewer**, la cual tiene variantes tanto para valores discretos como para valores continuos.

Vamos a probar la paleta "Dark2" de ColorBrewer, que es una paleta *qualitativa*. Cómo estamos modificando el *color*, la función a usar será `scale_color_brewer()`:

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line(aes(color = region)) +
  scale_color_brewer(type = "qual", palette = "Dark2")
```



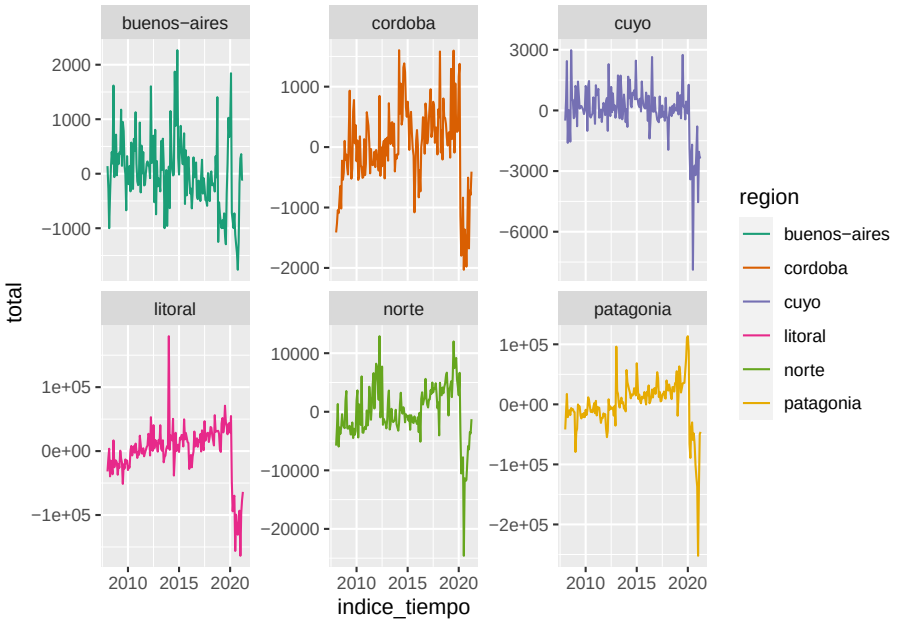
Desafío

A modo de prueba, cambia la paleta de colores actual por la de Viridis. Para eso tenés que usar `scale_color_viridis_d()`. La "d" del final viene de *discrete* y se usa para variables discretas o categorías, mientas que si los datos son continuos, se usa "c".

Como las variaciones en los visitantes de Patagonia y el Litoral son mucho más grandes en magnitud que las del resto de las regiones, éstas últimas aparecen casi como una línea recta constante en cero. Este es un problema que ninguna escala de colores puede solucionar.

Si queremos mostrar líneas de tiempo con escalas muy distintas, hay que usar paneles con escalas libres. Esto se hace poniendo `scales = "free_y"` en `facet_wrap()` para especificar que cada panel tiene su propia escala del eje vertical.

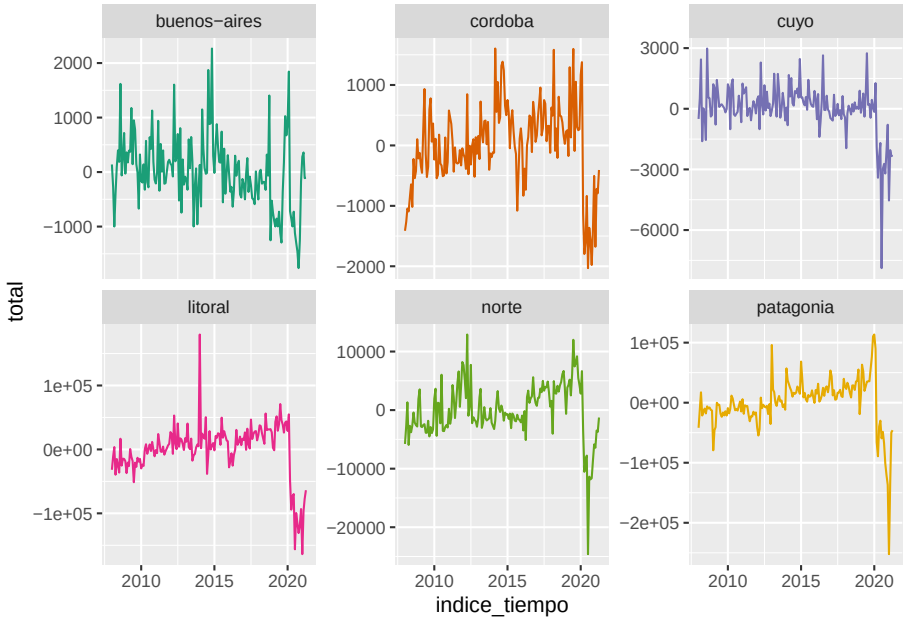
```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line(aes(color = region)) +
  scale_color_brewer(type = "qual", palette = "Dark2") +
  facet_wrap(~region, scales = "free_y")
```



Esto permite ver la variabilidad existente en todas las regiones independientemente de su escala vertical. Al mismo tiempo ahora los paneles son son fácilmente comparable entre si porque el rango del eje y es distinto en cada uno.

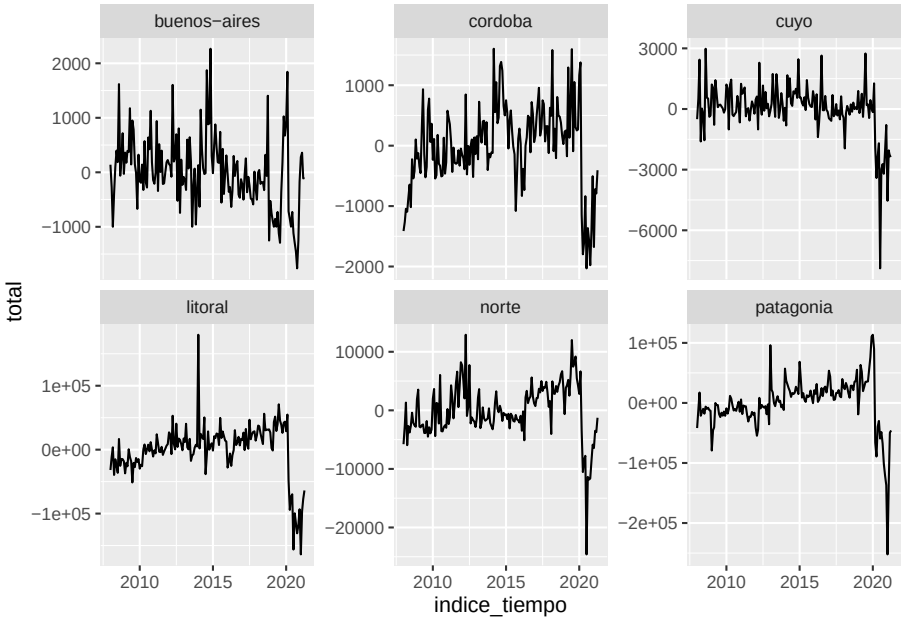
Como cada región está en su panel propio, ahora el color está codificando información redundante. Llegamos a un momento importante en el proceso de creación de un gráfico. Además de agregar capas, escalas y colores, también es importante saber cuándo **sacar**. En este caso, podemos sacar la leyenda de colores agregando `guide = "none"` a `scale_color_brewer()`.

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line(aes(color = region)) +
  scale_color_brewer(type = "qual", palette = "Dark2", guide = "none") +
  facet_wrap(~region, scales = "free_y")
```



Pero más radical es directamente sacar el mapeo del color. A veces la mejor escala de colores es sin escala de colores.

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line() +
  facet_wrap(~region, scales = "free_y")
```

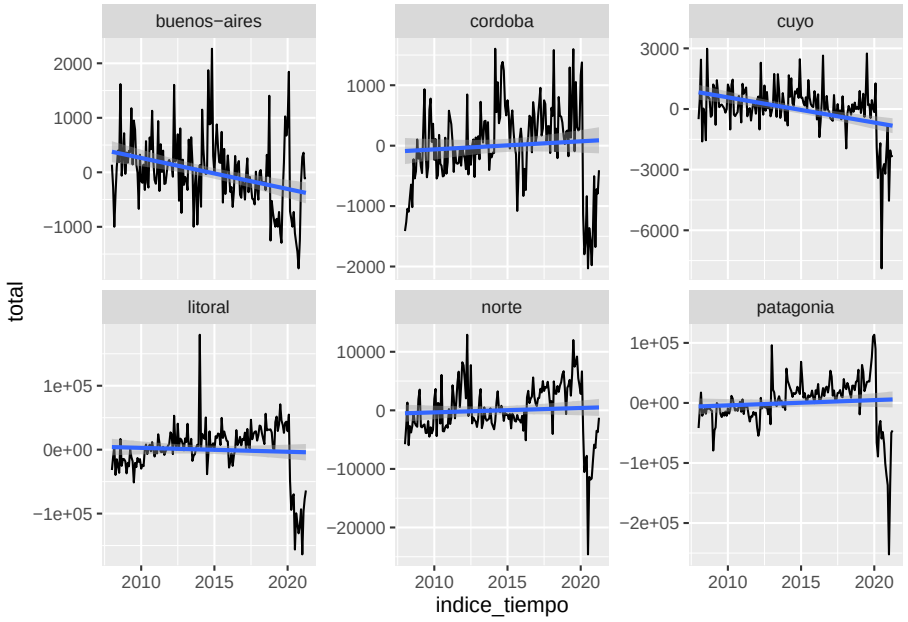


Ahora que cada región tiene su propio panel, se pueden ver las tendencias a largo plazo fácilmente. En casi todas las regiones la cantidad de visitantes estuvo en crecimiento desde 2008 y luego colapsó con la pandemia. Buenos Aires y Cuyo parecen ser la excepción; ambas regiones estaban estables antes de la pandemia.

Una buena forma de resaltar la tendencia a largo plazo es agregando una recta de regresión lineal. Vimos que una forma rápida de agregar una línea de tendencia es agregando una capa con la función `geom_smooth(method = "lm")`

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line() +
  geom_smooth(method = "lm") +
  facet_wrap(~region, scales = "free_y")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



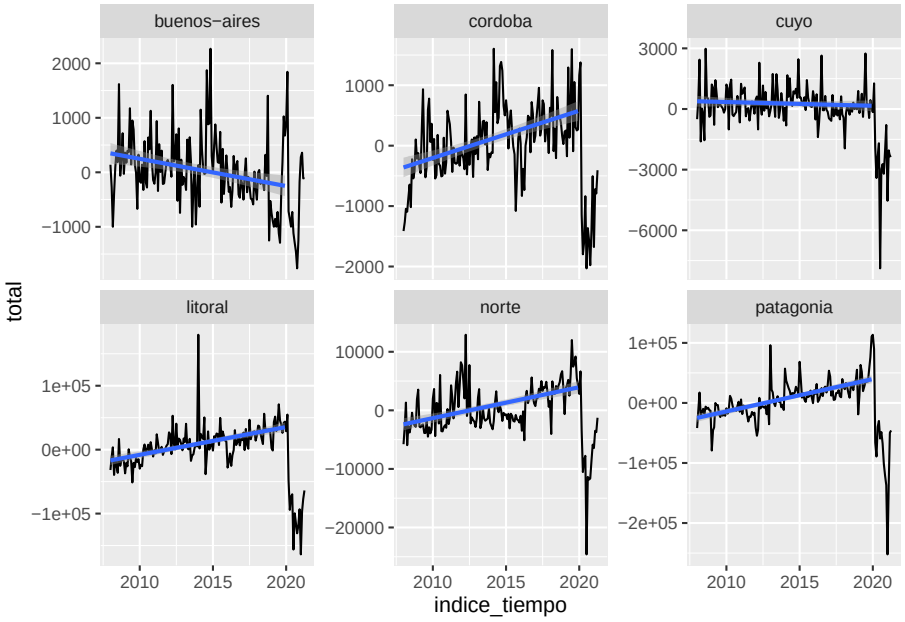
Pero esto tiene un problema. El colapso de los visitantes en 2020 y 2021 están generando una línea de tendencia chata o incluso negativa. ¿Cómo hacer para calcular la línea de tendencia usando sólo los datos anteriores a 2020?

Lo que se puede hacer es generar una variable extra con los datos pre-pandemia y luego usar esos datos en el `geom_smooth()`:

```
pre_pandemia <- filter(parques, lubridate::year(indice_tiempo) < 2020)

parques %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line() +
  geom_smooth(data = pre_pandemia,
              method = "lm") +
  facet_wrap(~region, scales = "free_y")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



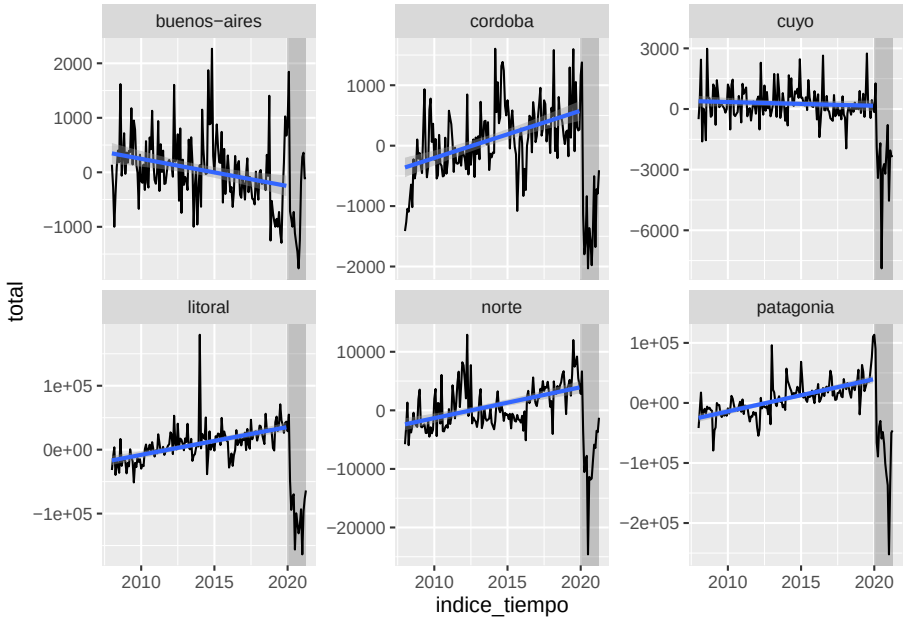
Del código anterior surge algo muy importante: es posible generar capas en un gráfico usando una tabla *distinta* a la que usamos para graficar las capas anteriores. Esto es útil principalmente para definir etiquetas o resaltar determinadas observaciones.

Estamos computando la regresión para los datos anteriores a 2020 porque los posteriores son anómalos. Sería buena idea indicar en el gráfico dónde empiezan los datos anómalos con un recuadro grisado.

La tarea de "anotar" un gráfico con geometrías que no hacen referencia a los datos se hace con la función `annotate()`. Toma el nombre de un geom y luego los parámetros estéticos **fuera** del `aes()`.

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  annotate("rect",
           xmin = as.Date("2020-01-01"), xmax = as.Date("2021-04-01"),
           ymin = -Inf, ymax = Inf,
           alpha = 0.3) +
  geom_line() +
  geom_smooth(data = pre_pandemia,
              method = "lm") +
  facet_wrap(~region, scales = "free_y")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



En este código, `annotate()` dibuja un rectángulo (el equivalente a `geom_rect()`) donde el límite izquierdo es el primero de enero de 2020, y el derecho es el primero de abril de 2021. Los límites superiores e inferiores son -Inf y +Inf, lo que indican que tienen que cubrir todos los límites del gráfico. Además, con `alpha = 0.3`, el rectángulo es semitransparente.

9.1.2. Escala de ejes

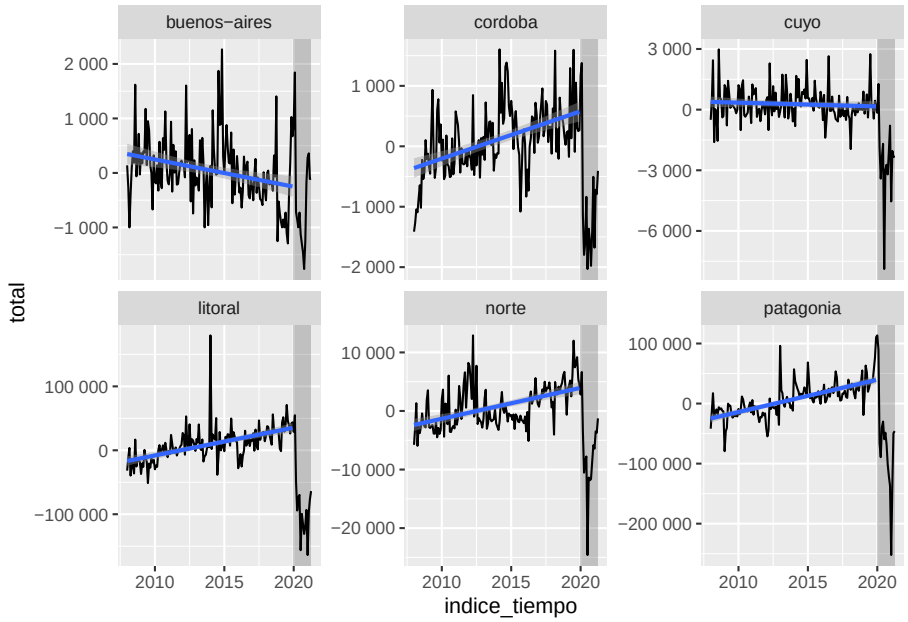
Además de modificar escalas de colores, también se pueden modificar las escalas de posición, es decir, los ejes.

En este gráfico, los números del eje y son poco amigables; especialmente la notación científica para los paneles de litoral y de patagonia. Entonces vamos a modificar el eje y agregando una capa con `scale_y_continuous()` (porque `total` es una variable continua) usando el argumento `labels` de esa función.

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  annotate("rect",
           xmin = as.Date("2020-01-01"), xmax = as.Date("2021-04-01"),
           ymin = -Inf, ymax = Inf,
           alpha = 0.3) +
  geom_line() +
  geom_smooth(data = pre_pandemia,
```

```
              method = "lm") +
  facet_wrap(~region, scales = "free_y") +
  scale_y_continuous(labels = scales::number_format())
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Este código usa la función `nombre_format()` del paquete **scales**. Este es un paquete que no hace falta instalar por separado porque viene con ggplot2 e implementa transformaciones de escala y funciones para dar formato. La función `number_format()` devuelve una función (sí, es una función que devuelve una función) que se encarga de darle un formato bonito a los números, incluyendo redondeo de cifras significativas, separador de miles o de decimales, etc.

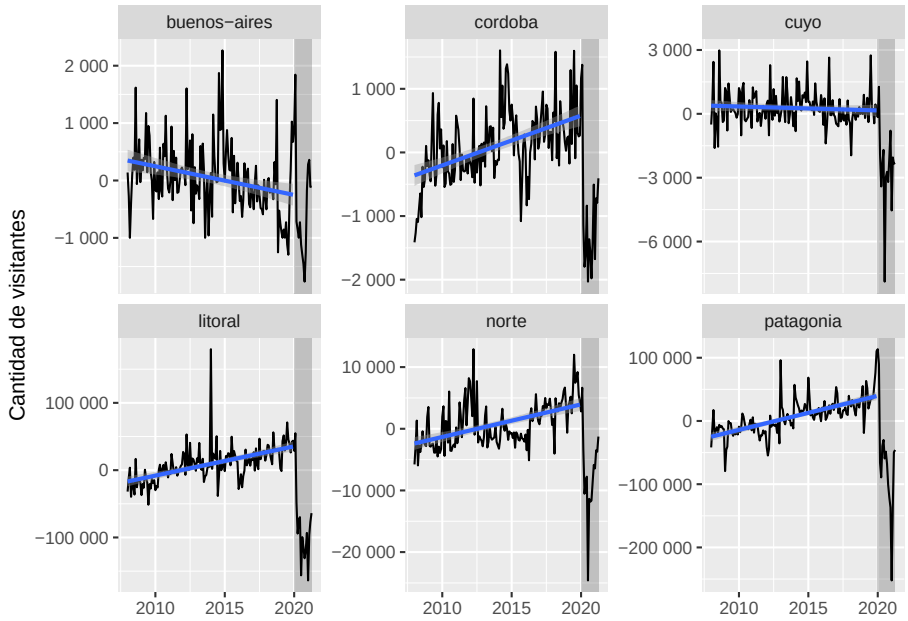
9.1.3. Etiquetas y texto

Algo que hay que arreglar sí o sí en este gráfico antes publicarlo es el nombre de los ejes y los paneles. Primero, cambiemos las etiquetas de los ejes agregando una nueva capa con la función `labs()`.

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  annotate("rect",
```

```
          xmin = as.Date("2020-01-01"), xmax = as.Date("2021-04-01"),
          ymin = -Inf, ymax = Inf,
          alpha = 0.3) +
  geom_line() +
  geom_smooth(data = pre_pandemia,
              method = "lm") +
  facet_wrap(~region, scales = "free_y") +
  scale_y_continuous(labels = scales::number_format()) +
  labs(y = "Cantidad de visitantes",
       x = NULL)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Este código le asigna el texto "Cantidad de visitantes" al eje vertical, y le quita la etiqueta al eje x, ya que se sobreentiende por convención que se trata del eje de tiempo.

Para cambiar el nombre de los paneles hay que primero generar un vector que asigne cada nombre "de computadora" (o sea, sin espacios ni mayúsculas ni tildes) al nombre "para personas". La forma de hacer esto es creando un vector con nombres.

```
etiquetas_regiones <- c("buenos-aires" = "Buenos Aires",
                        "cordoba"      = "Córdoba",
                        "cuyo"         = "Cuyo",
                        "litoral"      = "Litoral",
                        "norte"        = "Norte",
                        "patagonia"    = "Patagonia")
etiquetas_regiones
```

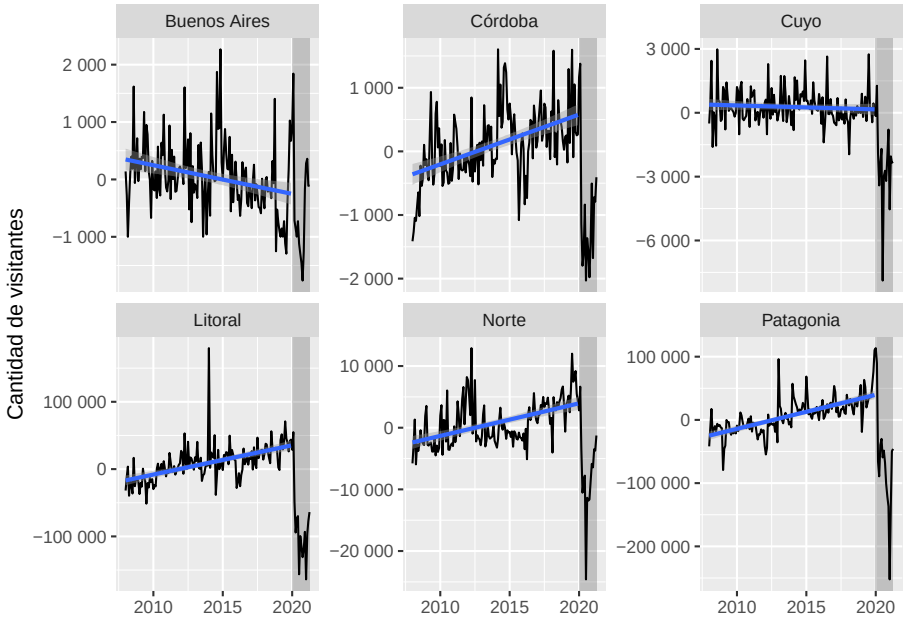
```
##   buenos-aires        cordoba           cuyo        litoral          norte
## "Buenos Aires"      "Córdoba"         "Cuyo"      "Litoral"        "Norte"
##      patagonia
##    "Patagonia"
```

Fijate que el contenido del vector son las regiones como queremos que se muestren en el gráfico, y luego cada elemento tiene como nombre el texto como está en los datos.

Ahora podemos usar ese vector como un "labeller". En jerga de ggplot2, los "labellers" son funciones que "etiquetan" los paneles.

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  annotate("rect",
           xmin = as.Date("2020-01-01"), xmax = as.Date("2021-04-01"),
           ymin = -Inf, ymax = Inf,
           alpha = 0.3) +
  geom_line() +
  geom_smooth(data = pre_pandemia,
              method = "lm") +
  facet_wrap(~region, scales = "free_y",
             labeller = labeller(region = etiquetas_regiones)) +
  scale_y_continuous(labels = scales::number_format()) +
  labs(y = "Cantidad de visitantes",
       x = NULL)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

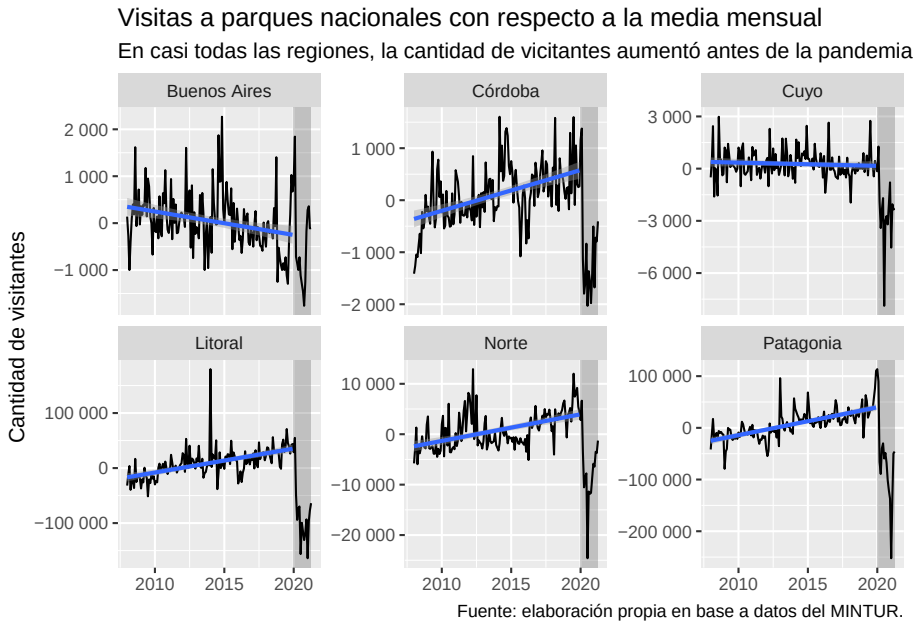


Si el gráfico va a circular por distintos lugares o fuera de su contexto inicial, conviene sumar algún texto que describa el gráfico y la fuente de datos.

Esto se puede lograr agregando título, subtítulo y caption (mejor conocido como epígrafe) dentro de la función `labs()`.

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  annotate("rect",
           xmin = as.Date("2020-01-01"), xmax = as.Date("2021-04-01"),
           ymin = -Inf, ymax = Inf,
           alpha = 0.3) +
  geom_line() +
  geom_smooth(data = pre_pandemia,
              method = "lm") +
  facet_wrap(~region, scales = "free_y",
             labeller = labeller(region = etiquetas_regiones)) +
  scale_y_continuous(labels = scales::number_format()) +
  labs(y = "Cantidad de visitantes",
       x = NULL,
       title = "Visitas a parques nacionales con respecto a la media mensual",
       subtitle = "En casi todas las regiones, la cantidad de vicitantes aumentó antes
       caption = "Fuente: elaboración propia en base a datos del MINTUR.")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



9.2. Temas

Hasta ahora llegamos a un gráfico muy funcional. Muestra los datos, resalta las características más importantes y tiene etiquetas legibles. Pero es medio aburrido. Peor, al tener los colores por defecto de ggplot2, no tiene ningún toque personal que lo haga resaltar. Entonces, lo último que queda por hacer es cambiar la apariencia global del gráfico usando los *temas*.

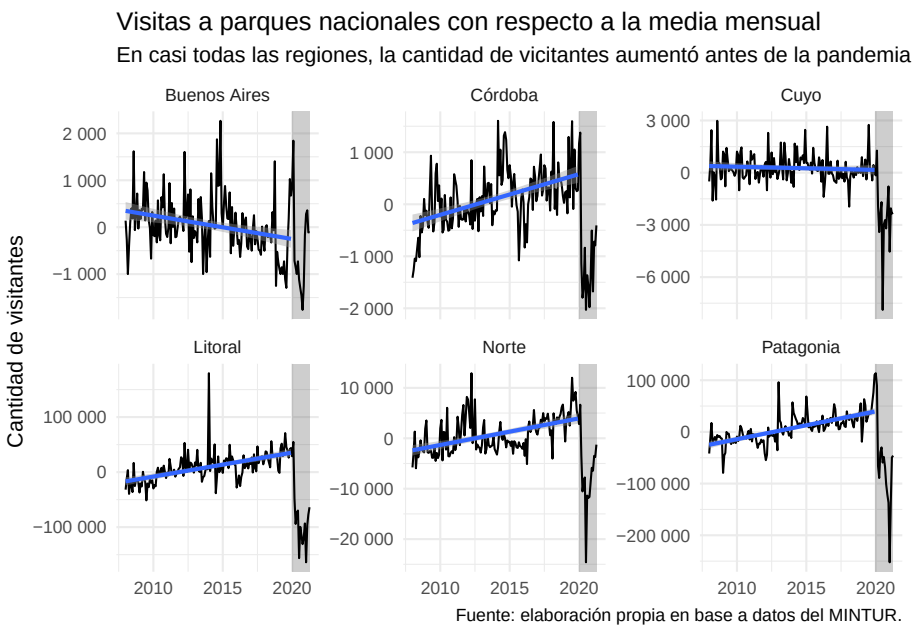
ggplot2 tiene muchos *temas* disponibles y para todos los gustos. Además hay otros paquetes que extienden las posibilidades, por ejemplo {ggthemes}.

Por defecto ggplot2 usa `theme_grey()`, probemos `theme_minimal()`.

```
parques %>%
  ggplot(aes(indice_tiempo, total)) +
  annotate("rect",
           xmin = as.Date("2020-01-01"), xmax = as.Date("2021-04-01"),
           ymin = -Inf, ymax = Inf,
           alpha = 0.3) +
  geom_line() +
  geom_smooth(data = pre_pandemia,
              method = "lm") +
  facet_wrap(~region, scales = "free_y",
             labeller = labeller(region = etiquetas_regiones)) +
```

```
  scale_y_continuous(labels = scales::number_format()) +
  labs(y = "Cantidad de visitantes",
       x = NULL,
       title = "Visitas a parques nacionales con respecto a la media mensual",
       subtitle = "En casi todas las regiones, la cantidad de vicitantes aumentó antes
       caption = "Fuente: elaboración propia en base a datos del MINTUR.") +
  theme_minimal()
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Ahora es tu turno. Elegí un tema que te guste y probalo. Además, si se te ocurre algún título mejor modificalo!

Junto con las funciones `theme_...()`, hay una función llamada `theme()` que permite cambiar la apariencia de cualquier elemento del gráfico. Tiene casi infinitas opciones y si algún momento te desvelas intentando cambiar esa línea o ese borde, seguro que `theme()` tiene alguna opción para hacer eso.

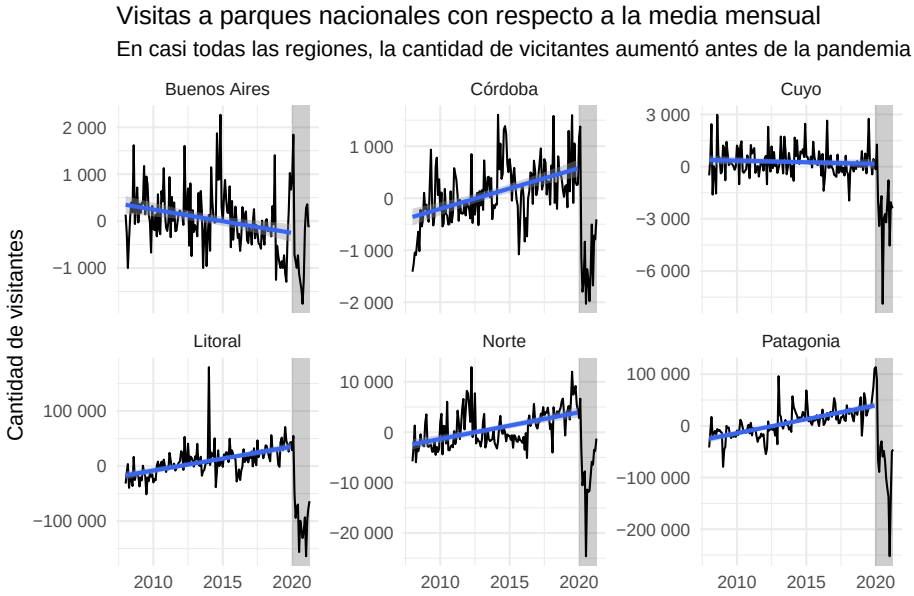
Finalmente, así quedó el código que genera el gráfico final.

```
parques <- readr::read_csv("datos/parques_tidy.csv") %>%
  group_by(mes = lubridate::month(indice_tiempo), region) %>%
  mutate(total = total - mean(total)) %>%
  ungroup()
```

```r
pre_pandemia <- filter(parques, lubridate::year(indice_tiempo) < 2020)

etiquetas_regiones <- c("buenos-aires" = "Buenos Aires",
                        "cordoba"      = "Córdoba",
                        "cuyo"         = "Cuyo",
                        "litoral"      = "Litoral",
                        "norte"        = "Norte",
                        "patagonia"    = "Patagonia")

parques %>%
  ggplot(aes(indice_tiempo, total)) +
  annotate("rect",
           xmin = as.Date("2020-01-01"), xmax = as.Date("2021-04-01"),
           ymin = -Inf, ymax = Inf,
           alpha = 0.3) +
  geom_line() +
  geom_smooth(data = pre_pandemia,
              method = "lm") +
  facet_wrap(~region, scales = "free_y",
             labeller = labeller(region = etiquetas_regiones)) +
  scale_y_continuous(labels = scales::number_format()) +
  labs(y = "Cantidad de visitantes",
       x = NULL,
       title = "Visitas a parques nacionales con respecto a la media mensual",
       subtitle = "En casi todas las regiones, la cantidad de vicitantes aumentó antes de la pand
       caption = "Fuente: elaboración propia en base a datos del MINTUR.") +
  theme_minimal()
```


Visitas a parques nacionales con respecto a la media mensual
En casi todas las regiones, la cantidad de vicitantes aumentó antes de la pandemia
Buenos Aires
Córdoba
Cuyo
Litoral
Norte
Patagonia
Cantidad de visitantes
2 000
1 000
0
−1 000
−2 000
3 000
−3 000
−6 000
100 000
−100 000
10 000
−10 000
−20 000
−200 000
2010
2015
2020
Fuente: elaboración propia en base a datos del MINTUR.

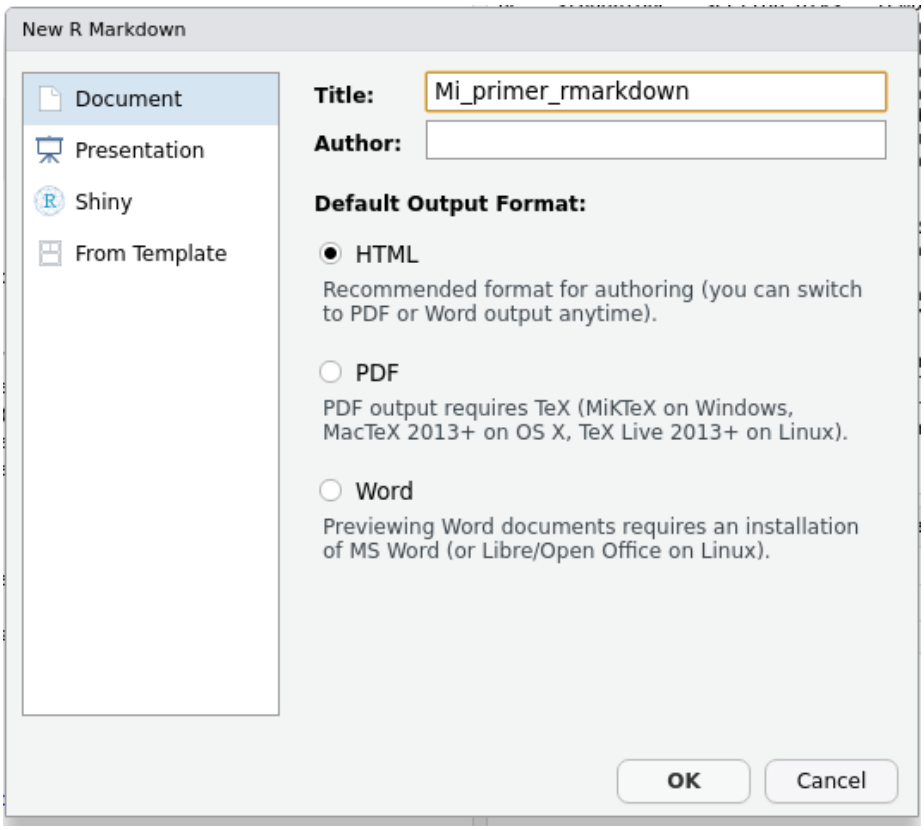
Capítulo 10

Informes reproducibles

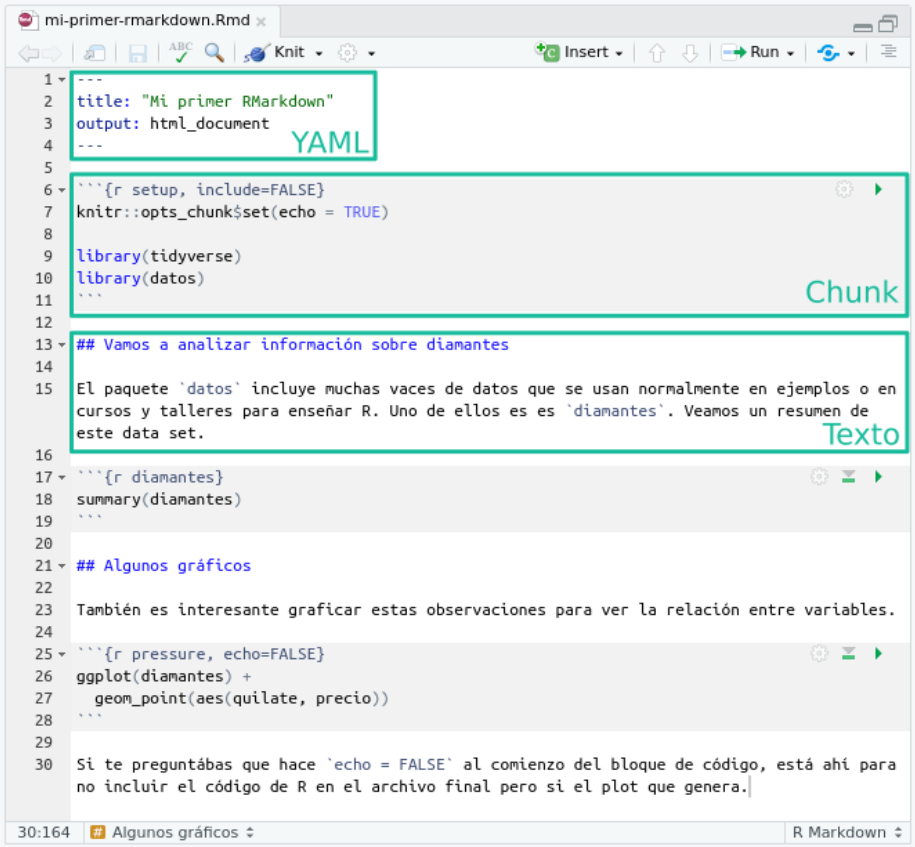
Durante este curso fuiste creando varios documentos con R Markdown en HTML. Este es un formato que tiene un montón de flexibilidad, pero seguramente no es el único que necesitás. Casi seguro que los informes los tengas que presentar en formato PDF o, incluso, ¡en papel impreso! RMarkdown, y todo un amplio ecosistema de otros paquetes, permite generar documentos en múltiples formatos usando el mismo archivo de texto plano.

10.1. Eligiendo el formato de salida

Ya habrás visto esto cuando creás un archivo markdown nuevo, RStudio te permite elegir entre tres formatos de salida:



Cuál es el formato de salida de un archivo de R Markdown se determina principalmente con la opción `output` en el encabezado yaml. Si mirás el encabezado del archivo R Markdown de ejemplo vas a ver que la opción `output` dice `html_document`.



Ese `html_document` no es otra cosa que una función de rmarkdown llamada `html_document`. Como te podrás imaginar, rmarkdown tiene una serie de otras funciones que definen formatos de salida. Los dos que seguramente te van a servir más son `pdf_document` y `word_document` que justamente generan PDFs y archivos de Word, respectivamente.

Para crear un documento de R Markdown que genere un archivo PDF basta con cambiar el `output` en el encabezado por esto:

```
---
output: pdf_document
---
```

Para que el documento se genere correctamente hace falta instalar LaTeX, que es un sistema de composición de textos. Aunque parezca mentira, la mejor forma de instalar LaTeX para usar R Markdown es instalando el paquete {tinytex} con `install.packages("tinytex")` y luego correr `tinytex::install_tinytex()`. Esto va a instalar una versión pequeña de LaTeX en un lugar donde luego rmarkdown lo puede usar. Esta es la forma altamente recomendada para generar

PDFs con R Markdown que va a evitarte un montón de dolores de cabeza.

Análogamente, podés generar un archivo de word cambiando el `output` así:

```
---
output: word_document
---
```

Y ya está. En la gran mayoría de los casos no vas a tener que modificar nada más del código ni el texto.

Desafío

Agarrá el reporte que estuviste armando en los desafíos o alguno que usaste durante el curso y compilalo en PDF y luego en Word.

10.2. Personalizando la salida

Cada función de formato viene con sus opciones de personalización que podés acceder leyendo su documentación. Para ver la documentación de `html_document`, usa este comando:

```
?rmarkdown::html_document
```

Vas a ver que tiene un montón de argumentos que modifican la salida. La forma de setear estos argumentos en un documento de R Markdown es, de nuevo, en el encabezado. Cada argumento de la función de salida (`html_document` en este caso) es un elemento debajo de la función de output.

Por ejemplo, para que un documento de html tenga una tabla de contenidos hay que setear el argumento `toc` (de **t**able **o**f **c**ontents) a `TRUE`. En el encabezado, esto queda así:

```
---
output:
  html_document:
    toc: TRUE
---
```

Conviene mirar eso con un poco de detenimiento porque requiere "traducir" código de R –cual los argumentos de una función se fijan entre paréntesis y con `=`– en código de yaml –donde los argumentos de la función son una lista cuyos elementos se definen con `:`.

En R lo que vemos como `html_document(toc = TRUE)` se traduce a yaml como

```
html_document:
  toc: TRUE
```

Si vas a la ayuda de `pdf_document` vas a ver que también tiene un argumento llamado `toc`. Algunos argumentos son compartidos, lo cual hace que se aún más fácil generar un mismo reporte en muchos formatos haciendo muy pocos cambios.

Una forma rápida de hacer tus informes más vistosos es cambiarle el tema visual. `html_document` permite elegir entre una serie de temas usando el argumento `theme`. Por ejemplo, poniendo esto en el encabezado, generás un documento HTML con un fondo oscuro

```
output:
  html_document:
    toc: TRUE
    theme: darkly
```

Desafío

Andá a la ayuda de `html_document` y fijate cuáles son los valores válidos para el argumento `theme`. ¡Probá algunos!

10.3. Reportes parametrizados

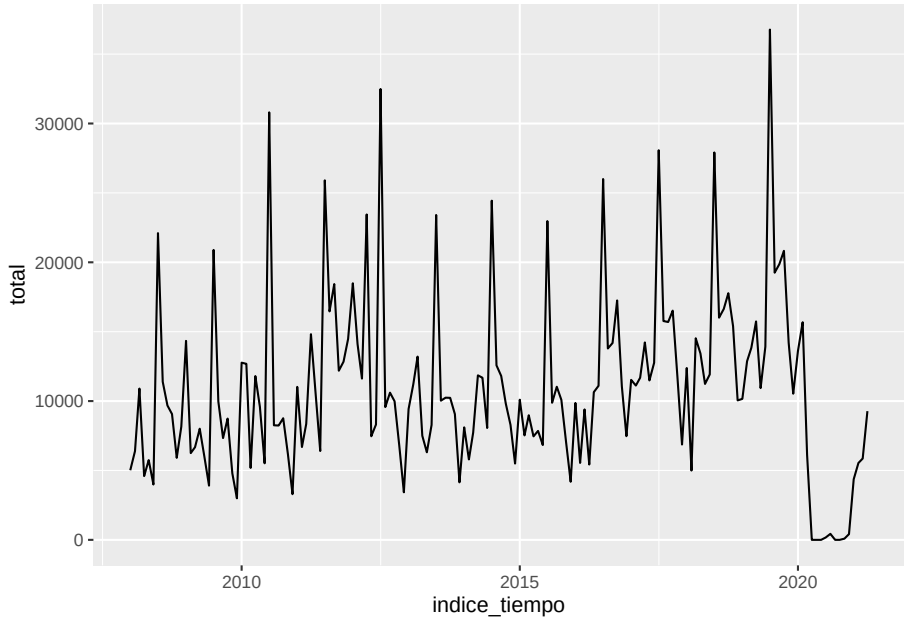
Es muy común tener que hacer un reporte cuyo resultado dependa de ciertos parámetros.

Por ejemplo, podrías tener un reporte que analiza la evolución de visitantes en parques nacionales en una determinada región de Argentina con el siguiente código:

```
library(readr)
library(dplyr)
library(ggplot2)

parques <- read_csv("datos/parques_tidy.csv")

parques %>%
  filter(region == "norte") %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line()
```



Si ahora querés hacer el mismo reporte pero para la región de Patagonia, tenés que abrir el archivo y modificar la llamada a `filter` para quedarte sólo con esa región:

```
library(readr)
library(dplyr)
library(ggplot2)

parques <- read_csv("datos/parques_tidy.csv")

parques %>%
  filter(region == "patagonia") %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line()
```

Si el reporte es largo y usa el nombre de la región en múltiples lugares cambiar "norte" por "patagonia" puede ser tedioso y propenso a error, ya que te obliga a modificar muchas partes del código. Y si después tenés que hacer el mismo reporte para "cordoba"...

En estas situaciones podés crear un reporte parametrizado. La idea es que el reporte tiene una serie de parámetros que puede modificar la salida. Es como si el archivo de R Markdown fuera una gran función con sus argumentos!

Para generar un reporte parametrizado hay que agregar un elemento llamado `params` al encabezado con la lista de parámetros y sus valores por default.

```
params:
  region: norte
```

Luego, en el código de R vas a tener acceso a una variable llamada `params` que es una lista que contiene los parámetros y su valor. Para acceder al valor de cada parámetros se usa el operador `$` de la siguiente manera:

```
params$region
```

```
## [1] "norte"
```

De esta manera, el código original se puede modificar para usar el valor de la región almacenado en `params$region`

```
library(readr)
library(dplyr)
library(ggplot2)

parques <- read_csv("datos/parques_tidy.csv")

parques %>%
  filter(region == params$region) %>%
  ggplot(aes(indice_tiempo, total)) +
  geom_line()
```

Y ahora el mismo código puede funcionar para distintas regiones. Para crear reportes distintos para cada región sólo hay que modificar el valor del parámetro en el encabezado:

```
params:
  pais: patagonia
```

Desafío

Agregá al menos un parámetro al reporte que venís armando.

10.4. Control de chunks

En la sección Introducción a R Markdown vimos que un chunk tiene una pinta como esta:

```
```{r nombre-del-chunk}

```
```

Ponerle nombre al chunk no es obligatorio pero está bueno para tener una idea de qué hace cada uno, lo cual se vuelve más importante a medida que un reporte se vuelve más largo y complejo. Pero lo que no dijimos es que además del nombre, entre las llave se pueden poner un montón de opciones que cambian el comportamiento y la apariencia del resultado del chunk.

Para cambiar las opciones de un chunk, lo único que hay que hacer es listarlas dentro de los corchetes. Por ejemplo:

````
```{r nombre-del-chunk, echo = FALSE, message = FALSE}

```
````

Hay una serie de opciones particularmente importante es la que controla si el código se ejecuta y si el resultado del código se va a mostrar en el reporte o no:

- `eval = FALSE` evita que se ejecute el código del chunk, de manera que tampoco va a mostrar resultados. Es útil para mostrar códigos de ejemplo si estás escribiendo, por ejemplo un documento para enseñar R.
- `echo = FALSE` ejecuta el código del chunk y muestra los resultados, pero oculta el código en el reporte. Esto es útil para escribir reportes para personas que no necesitan ver el código de R que generó el gráfico o tabla que querés mostrar.
- `include = FALSE` corre el código pero oculta tanto el código como los resultados. Es útil para usar en chunks de configuración general donde, por ejemplo, cargas las librerías.

Si estás escribiendo un informe en el que no querés que se muestre ningún código, agregarle `echo = FALSE` a cada chunk nuevo se vuelve tedioso. La solución es cambiar la opción de forma global de manera que aplique a todos los chunks. Esto se hace mediante la función `knitr::opts_chunk$set()`, que setea las opciones globales de los chunks que le siguen. Si querés que todos los chunks tengan `echo = TRUE` crearías un chunk así:

````
```{r setup, include = FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
````

Generalmente tiene sentido poner esto en el primer chunk de un documento, que como suele ser cuestiones de configuración del reporte, también conviene ponerle `include = FALSE`.

Habrás visto que a veces algunas funciones devuelven mensajes sobre lo que hacen. Por ejemplo, cuando `read_csv` lee un archivo describe el tipo de dato de cada columna:

```
parques <- read_csv("datos/parques.csv")
```

```
## Rows: 160 Columns: 22
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (1): indice_tiempo
## dbl (21): residentes, no_residentes, total, buenos_aires_residentes, buenos_...
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Esto es útil cuando uno está haciendo trabajo interactivo pero en general no quiere que quede en el reporte. Para que no muestre estos mensajes basta con poner la opción `message = FALSE`

````
```{r message = FALSE}
paises <- read_csv("datos/paises.csv")
```
````

En general no pasa nada si ignorás los mensajes. Son cuestiones diagnósticas extras que sirven para que vos, como humano, te enteres de lo que hizo una función. Distinto son las advertencias, o "warnings". Una advertencia te está diciendo que hay algo "raro" en el código que puede significar que hay algo mal. No llega al nivel de error, que es algo que literalmente "no computa". Por ejemplo, `sqrt` tira una advertencia cuando recibe números negativos.

```
i <- sqrt(-1)
```

```
## Warning in sqrt(-1): NaNs produced
```

Si un chunk tira una advertencia que es esperable pero no querés que aparezca en el reporte, podés ocultarlas con la opción `warning = FALSE`.

````
```{r warning = FALSE}
i <- sqrt(-1)
```
````

Finalmente, una opción tan poderosa como peligrosa es `cache = TRUE`. Lo que hace es que en vez de correr el código de un chunk cada vez que *kniteás* el documento, guarda el resultado del chunk en el disco para reutilizar la próxima

vez que crees el reporte. Esto es muy cómo si en chunk tiene un código que tarda mucho en correr. Por ejemplo el siguiente chunk va a tardar 10 minutos en correr la primera vez que knitees el reporte, pero luego va a ser mucho más rápido:

````
```{r cache = FALSE}
datos <- funcion_que_tarda_10_minutos(x)
```
````

knitr es bastante inteligente y va a invalidar la cache si cambiás el código del chunk. Pero, ¿qué pasa si cambiás algo del código previo que cambia el valor de `x` o incluso el funcionamiento de `function_que_targa_10_minutos`? Es posible que knitr no se de cuenta y use la cache, con el resultado de que `datos` va a tener un valor incorrecto. Hay formas de decirle a knitr de qué depende cada chunk y así obtener una cache más "inteligente" pero es algo que se vuelve complicado muy rápido.

En resumen, es bueno usar la cache pero sólo cuando es imprescindible.

Apéndice A

Desafíos de práctica

En esta sección te proponemos desafíos y ejercicios para prácticar o revisar lo que vimos en los distintos capítulos. Cada una de las subsección estará asociada a un conjunto de capítulos y buscan relacionar los distintos temas y herramientas.

Desafío 1

Capítulos asociados: Instalación de R y Rstudio, Uso de proyectos, Reportes con RMarkdown y lectura de datos.

1. Seguí las instrucciones que encontrarás en el Apéndice B para instalar R y Rstudio. Tené en cuenta que la instalación en computadoras con Windows requieren algunos pasos extra.
2. Abrí RStudio y creá tu primer proyecto. Este proyecto te acompañará a lo largo de todos los desafíos y ejercicios de este libro, por lo que va a necesitar un nombre fácil de recordar. Podés revisar las instrucciones acá.
3. Ahora es momento de instalar algunos paquetes para poder usar luego. Por ahora instalemos readr, remotes y datos.

```
install.packages("readr")
install.packages("remotes")
remotes::install_github("cienciadedatos/datos")
```

La tercera linea, si bien distinta a las anteriores, también instala un paquete. La diferencia es que instala el paquete desde un repositorio de GitHub donde suelen estar los paquetes en desarrollo en vez de desde el repositorio oficial de R (CRAN).

4. Creá un nuevo archivo R Markdown que se llame "01-lectura.Rmd" desde File -> New File -> R Markdown. Si bien el archivo puede tener cualquier nombre, siempre que sea informativo, te proponemos nombrarlos como número-nombre para poder ordenarlos y que te resulte más fácil encontrarlo dentro del proyecto. Es posible que necesites darle permiso a RStudio para que instale nuevos paquetes asociados a R Markdown. Te va a aparecer la plantilla por defecto; borrá todas las líneas empezando por la que dice `## R Markdown` (inclusive) y guardá el archivo. (Para guardar tenés podés ir a File -> Save o hacer click en en el disquette).

5. Aprovechemos el archivo recién creado para leer datos. Te proponemos que revises el capítulo de lectura de datos y sigas las instrucciones para leer los datos.

6. ¡Listo! En la próxima sección practicaremos con esos y otros datos.

Desafío 2

Capítulos asociados: Manipulación de datos con dplyr

Armá un nuevo archivo de R Markdown, borrá todas las líneas empezando por la que dice `## R Markdown` (inclusive) y guardá el archivo con algún nombre descriptivo.

1. Creá un nuevo bloque de código y pegá estas líneas de código.

```
library(datos)
vuelos <- vuelos
```

Corré el bloque. Va a cargar una tabla de ejemplo que vienen en el paquete datos.

Escribí el código que resuelve cada uno de los siguientes puntos en su propio bloque.

1. Fijate qué es cada columna de la tabla vuelos. Escribí `?vuelos` en la consola.

2. Usando la función `filter()`, encontrá todos los vuelos que:

    a. Tuvieron un retraso de llegada de dos o más horas
    b. Volaron a Houston (IAH oHOU)
    c. Fueron operados por United, American o Delta
    d. Llegaron más de dos horas tarde, pero no salieron tarde
    e. Partieron entre la medianoche y las 6 a.m. (incluyente)

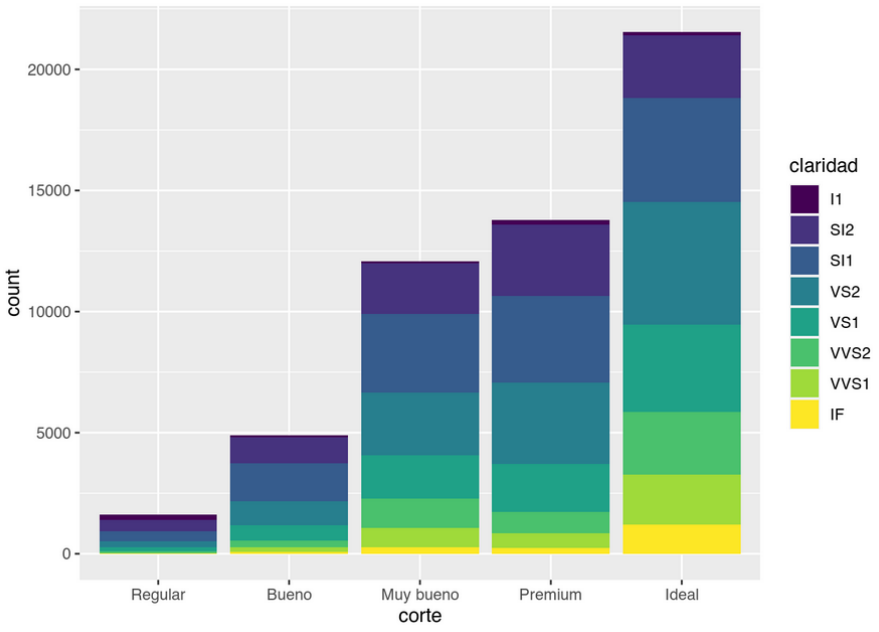
3. Las variables `horario_salida` y `salida_programada` tienen un formato conveniente para leer, pero es difícil realizar cualquier cálculo con ellas porque no son realmente números continuos. (Ayuda: para hacer la transformación hay que hacer `(horario %/% 100 * 60 + horario %% 100) %% 1440`)
4. Calculá el tiempo de vuelo usando `horario_llegada - horario_salida`. ¿Es igual a `tiempo_vuelo`? ¿Cuál es el problema?
5. Calculá el tiempo de vuelo promedio para cada aeropuerto de origen.
6. Calculá el retraso medio para cada aerolínea.
7. ¡Esta es la prueba de fuego! : Kniteá el documento haciendo click en el botón knitr. Si tu código es reproducible, tiene que ejecutar todo sin errores y devolver un documento html con el resultado.

Desafío 3

Capítulos asociados: Gráficos con ggplot2

Usando el dataset diamantes del paquete datos intenten replicar los gráficos que aparecen a continuación:

- Gráfico 1: Cantidad de diamantes por corte y claridad



* Gráfico 2: Proporción de diamantes por corte y claridad

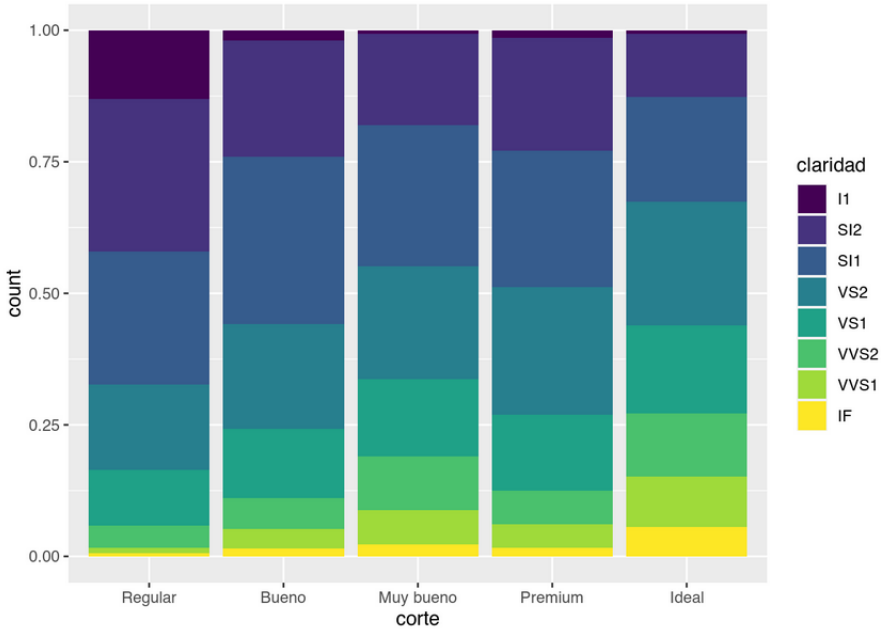


Figura A.1: Proporción por corte

- Gráfico 3: Cantidad de diamantes por corte y claridad

- Grafico 4: Replicar el siguiente gráfico

Este gráfico presenta los quilates en relación con el precio y la claridad. En el gráfico se marca el valor 5.01 que corresponde al máximo valor de quilates.

Desafío opcional

- Sobre el gráfico 1:

  - Agregar título: Cantidad de diamantes por tipo y claridad
  - Agregar como nota al pie: Fuente: conjunto de datos diamantes del paquete datos.
  - Cambiar los títulos de los ejes x, y y la leyenda. Nombres en español y con la primera letra en mayúsculas.

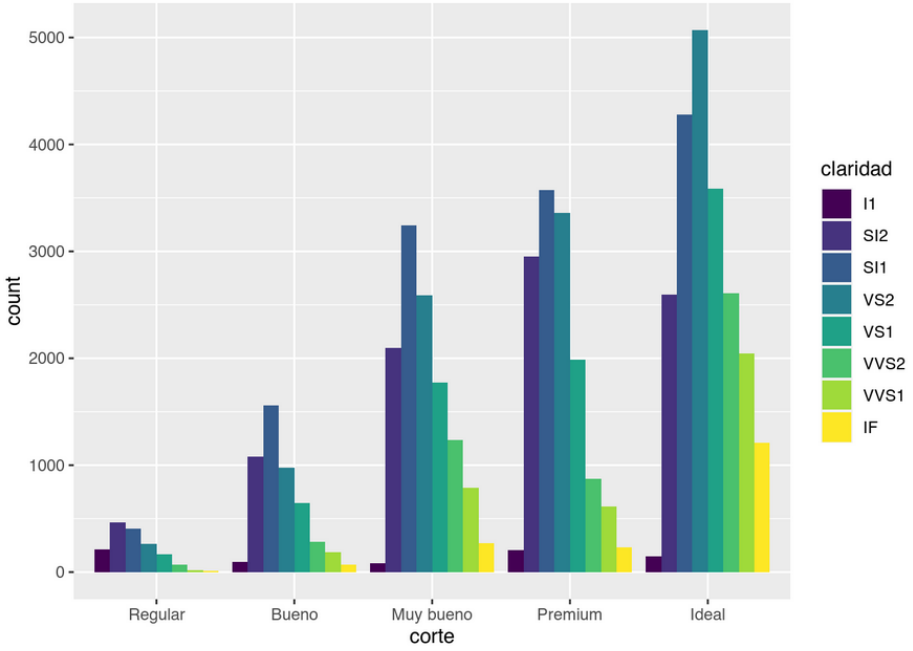


Figura A.2: Diamantes por corte

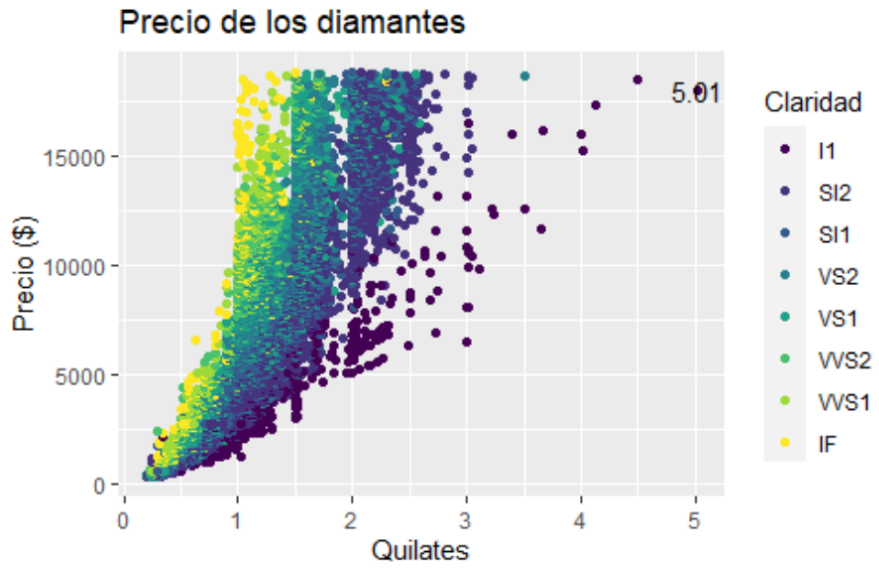


Figura A.3: Diamantes por corte

- Sobre el gráfico 2:
  - Agregar título: Diamantes por corte y claridad
  - Agregar subtítulo: proporciones de la claridad en cada tipo de corte.
  - Agregar como nota al pie: Fuente: conjunto de datos diamantes del paquete datos.
  - Cambiar los títulos de los ejes x, y y la leyenda. Nombres en español significativos y con la primera letra en mayúsculas.
- Sobre el gráfico 3:
  - Agregar título: Diamantes por corte y claridad
  - Agregar subtítulo: cantidades de acuerdo a la claridad en cada tipo de corte.
  - Agregar como nota al pie: Fuente: conjunto de datos diamantes del paquete datos.
  - Cambiar los títulos de los ejes x, y y la leyenda. Nombres en español significativos y con la primera letra en mayúsculas.

Desafío 4

Capítulos asociados: Tablas, gráficos e informes para publicar.

Llegaste al final del libro y con suerte al final de los desafíos. Es hora de generar el informe final para presentar todo lo que estuviste haciendo en este tiempo.

1. Abrí tu proyecto Abrí tu proyecto desde el explorador de archivos (haciendo doble click en el archivo .Rproj) o desde a interfaz de RStudio. Trabajá sobre el mismo Rmd, que ya va tomando forma.
2. Seleccioná los gráficos y/o tablas que generarte durante los desafíos o las clases y que quieran incluir en tu "informe final" y copia **todo** el código necesario para generarlos. Esto incluye el codigo que leer y modifica los datos.
3. Agregá títulos y mejorá las etiquetas en tus gráficos, si querés, también podés cambiar el tema o alguna escala para que tenga una apariencia distinta (tenés todo acá).

Si generaste tablas, podés mejorar su apariencia con alguna de las opciones de {kable} o {kableExtra} que vimos en el capítulode tablas.

4. Decidí en que formato vas a querer tu informe, html, word o pdf. ¿Necesita un indice?

Además hace las modificaciones que consideres necesarias para que tenga la apariencia que mejor se adapte a tus objetivos, ¿es necesario que se vea el código? ¿necesitás los mensajes que devuelve R?. Podés usar esté capítulo como guía.

Y finalmente… kniteá el documento!

Apéndice B

Instalando R y RStudio

En esta sección presentamos instrucciones de instalación de R y RStudio para Windows y derivados de Ubuntu. Notá que dado que tanto R como RStudio publican nuevas versiones periódicamente, es posible que estas instrucciones queden desactualizadas con el tiempo.

Instalando R

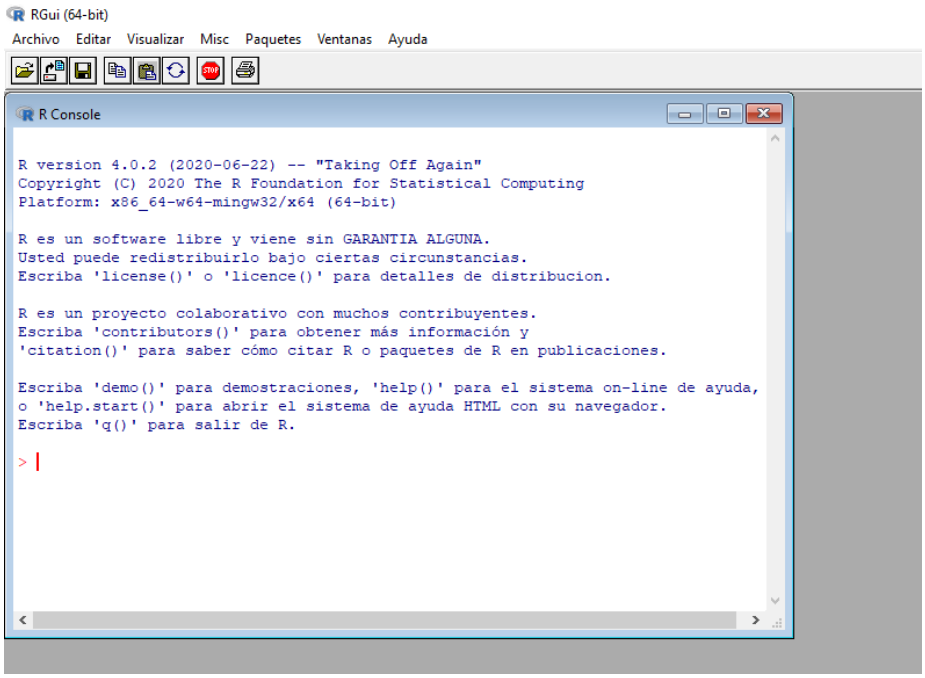
Windows

1. Entrá a https://cran.r-project.org/bin/windows/base/ y bajate el instalador haciendo click en el link grandote que dice "Download R x.x.x for Windows".

2. Una vez que se bajó, hacé doble click en el archivo y seguí las instrucciones del instalador.

Una vez que se termine de instalar, te va a aparecer un ícono como este en el escritorio o en los programas instalados:  .

Al ejecutarlo, les tiene que aparecer algo como esto:



Si ves una ventana así significa que ya tenés instalado R, pero seguí leyendo! **Todavía falta unos pasos** para poder sacarle todo el jugo.

Para instalar algunos paquetes de R vas a necesitar instalar un programa adicional llamado rtools.

1. Entrá a https://cran.r-project.org/bin/windows/Rtools/ y descargate el instalador en donde dice "On Windows 64-bit: rtoolsxx-x86_64.exe (recommended: includes both i386 and x64 compilers)"

2. Abrí la consola de R, poné esto y apretá enter:

```
writeLines('PATH="${RTOOLS40_HOME}\\usr\\bin;${PATH}"', con = "~/.Renviron")
```

3. Finalmente, para chequear que todo esté bien, cerrá R, volvé a abrirlo, escribí esto en la cosola y apretá enter:

```
Sys.which("make")
```

Debería salir algo como esto:

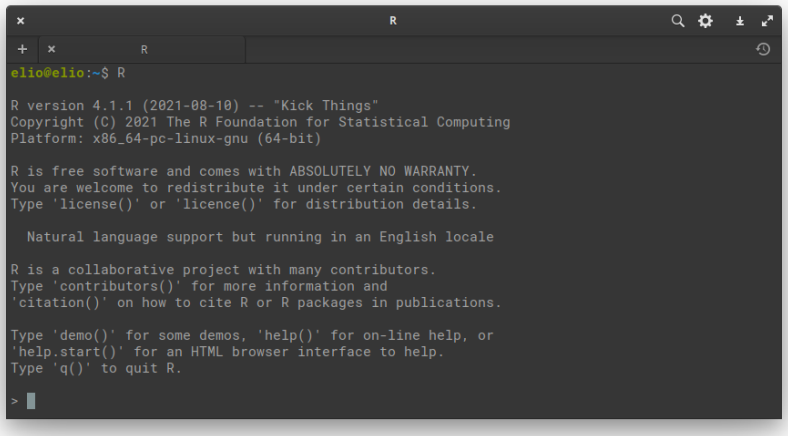
```
##                                make
## "C:\\rtools40\\usr\\bin\\make.exe"
```

Ubuntu o derivados

Para tener la última versión, tenés que agregar los repositorios de CRAN. Para hacerlo, vas a tener que tener permisos de administrador. Los detalles están en esta página, pero el resumen es:

```
sudo apt update -qq
sudo apt install --no-install-recommends software-properties-common dirmngr
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E298A3A825C0D65DFD57CBB65
sudo add-apt-repository "deb https://cloud.r-project.org/bin/linux/ubuntu $(lsb_release
sudo sudo apt install r-base r-base-dev
```

Si todo salió bien, tenés que tener instalado R en tu máquina y podés ejecutarlo con el comando `R` en la consola.

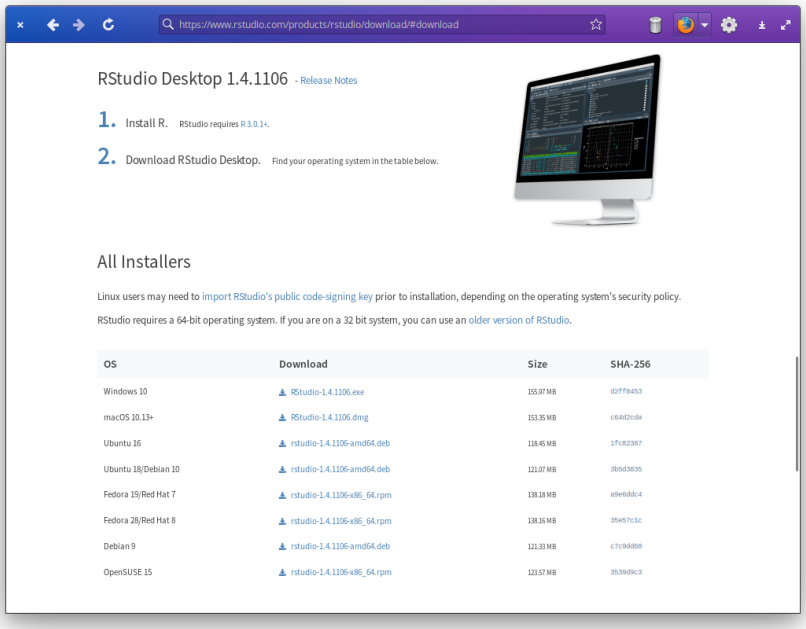


Mac

1. Ir a CRAN: https://cran.r-project.org
2. Hacer click en el link *"Download R for (Mac) OS X"*.
3. Descarga el paquete correspondiente a tu sistema operativo.
4. Ejecuta el paquete.

Instalando RStudio

Andá a https://www.rstudio.com/products/rstudio/download/#download. Abajo de todo está el listado de instaladores para cada plataforma; descargá la que corresponda a tu sistema operativo.



Windows

Como siempre, doble click en el archivo y seguir los pasos de instalación.

Ubuntu o derivados

Si tenés instalada una interfaz gráfica para instalar archivos .deb, ejecutando el archivo que descargaste ya vas a poder instalar RStudio. Si no, abrí una terminal en el directorio donde bajaste el archivo y ejecutá

```
sudo dpkg -i ARCHIVO
```

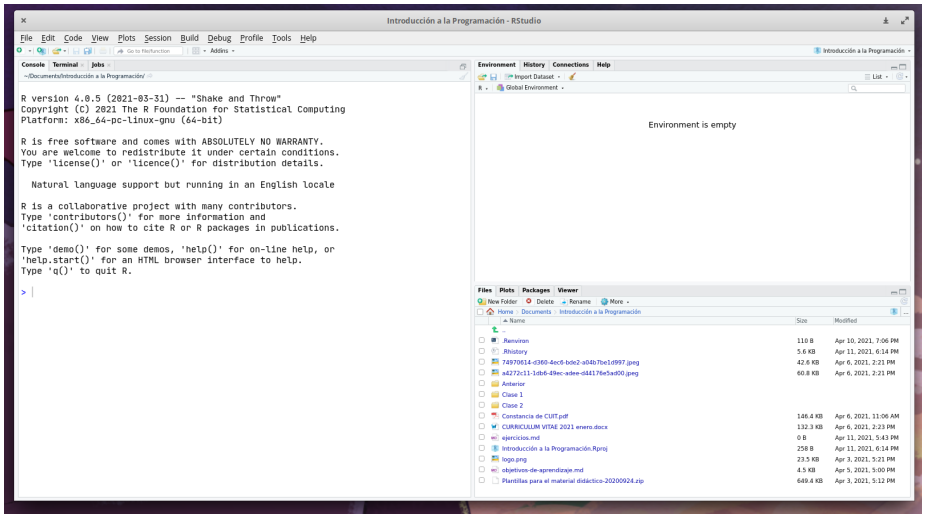
Reemplazando ARCHIVO por el nombre del archivo instalador.

Es posible que salte algún error por falta de alguna dependencia, en ese caso usá

```
sudo apt-get -f install
```

Y debería estar arreglado.

Una vez instalado, al ejecutar RStudio les tiene que aparecer una ventana como esta:



Mac

2. Descarga e instala RStudio: https://www.rstudio.com

La primera vez que abrimos RStudio nos ofrece instalar las herramientas de XCode para línea de comandos. Aceptamos haciendo clic en instalar.