LECTURE NOTES

ON

Foundations of Data Science

M.Tech (R18) I Semester

Prepared by

Mr. C Raghavendra

Assistant Professor

Computer Science and Engineering

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal- 500 043, Hyderabad.

UNIT - 1

Data science process

The data scientist is responsible for guiding a data science project from start to finish. Success in a data science project comes not from access to any one exotic tool, but from having quantifiable goals, good methodology, cross-discipline interactions, and a repeatable workflow.

The roles in a data science project:

ROLES AND RESPONSIBLITIES

Project sponsor : Represents the business interests; champions the project
Client : Represents end users' interests; domain expert
Data scientist : Sets and executes analytic strategy; communicates with sponsor and client
Data architect : Manages data and data storage; sometimes manages data collection
Operations : Manages infrastructure; deploys final project results

PROJECT SPONSOR:

The most important role in a data science project is the project sponsor. The sponsor is the person, who wants the data science result; generally they represent the business interests. The sponsor is responsible for deciding whether the project is a success or failure**.** Getting sponsor sign-off becomes the central organizing goal of a data science project.

CLIENT:

While the sponsor is the role that represents the business interest, the client is the role that represents the model's end users' interests. Sometimes the sponsor and client roles may be filled by the same person. Again, the data scientist may fill the client role if they can weight business trade-offs, but this isn't ideal.

The client is more hands-on than the sponsor; they're the interface between the technical details of building a good model and the day-to-day work process into which the model will be deployed.

DATA SCIENTIST

The next role in a data science project is the data scientist, who's responsible for taking all necessary steps to make steps, pick the data sources, and pick the tools to be used. Since they pick the techniques that will be tried, they have to be well informed about the project succeed, including setting the project strategy and keeping the client informed. They design the project statistics and machine learning. They're also responsible for project planning and tracking, though they may do this with a project management partner.

DATA ARCHITECT

The data architect is responsible for all of the data and its storage. Often this role is filled by someone outside of the data science group, such as a database administrator or architect. Data architects often manage data warehouses for many different projects, and they may only be available for quick consultation

OPERATIONS

The operations role is critical both in acquiring data and delivering the final results. The person filling this role usually has operational responsibilities outside of the data science group.

Stages of a data science project:

The ideal data science environment is one that encourages feedback and iteration between the data scientist and all other stakeholders. This is reflected in the lifecycle of a data science project.

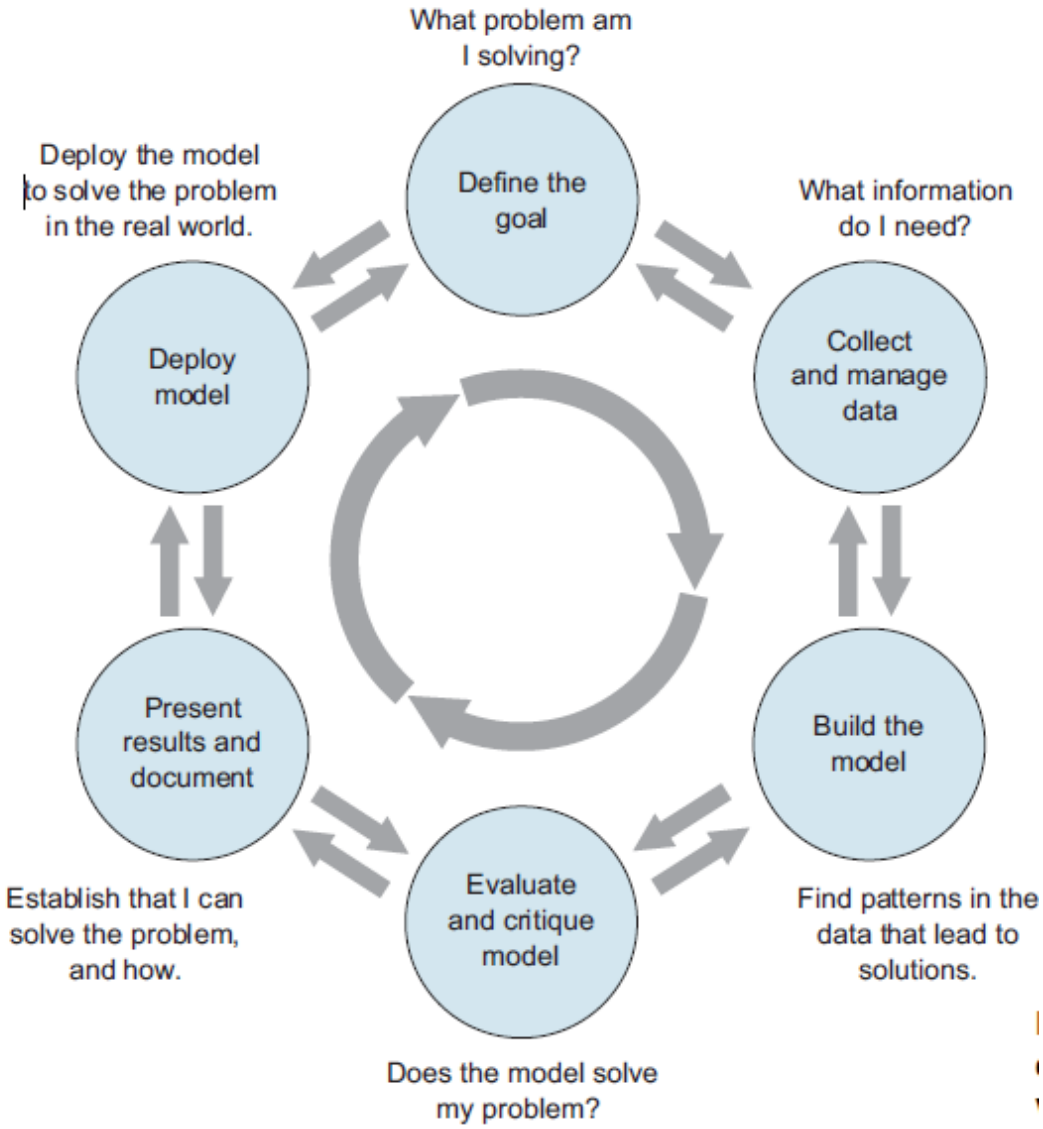


Fig. The life cycle of a data science project: loops within the loops

Defining the goal:

The first task in a data science project is to define a measurable and quantifiable goal. Why do the sponsors want the project in the first place? What do they lack, and what do they need? What are they doing to solve the problem now, and why isn't that good enough? What resources will you need: what kind of data and how much staff? Will you have domain experts to collaborate with, and what are the computational resources? How do the project sponsors plan to deploy your results? What are the constraints that have to be met for successful deployment?

Data collection and management

This step encompasses identifying the data you need, exploring it, and conditioning it to be suitable for analysis. This stage is often the most time-consuming step in the process

So far, we have dealt with small datasets that easily fit into your computer's memory. But what about datasets that are too large for your computer to handle as a whole? In this case, storing the

data outside of R and organizing it in a database is helpful. Connecting to the database allows you to retrieve only the chunks needed for the current analysis.

Even better, many large datasets are already available in public or private databases. You can query them without having to download the data first.

R can connect to almost any existing database type. Most common database types have R packages that allow you to connect to them (e.g., **RSQLite**, RMySQL, etc). Furthermore, the **dplyr** package you used in the previous chapter, in conjunction with **dbplyr** supports connecting to the widely-used open source databases sqlite, mysql and postgresql, as well as Google's bigquery, and it can also be extended to other database types (a vignette in the **dplyr** package explains how to do it). RStudio has created a website that provides documentation and best practices to work on database interfaces.

Interfacing with databases using **dplyr** focuses on retrieving and analyzing datasets by generating SELECTSQL statements, but it doesn't modify the database itself. **dplyr** does not offer functions to UPDATE or DELETEentries. If you need these functionalities, you will need to use additional R packages (e.g., **RSQLite**). Here we will demonstrate how to interact with a database using **dplyr**, using both the **dplyr**'s verb syntax and the SQL syntax.

The portal_mammals database

We will continue to explore the surveys data you are already familiar with from previous lessons. First, we are going to install the **dbplyr** package:

```
install.packages(c("dbplyr", "RSQLite"))
```

The SQLite database is contained in a single file portal_mammals.sqlite that you generated during the SQL lesson. If you don't have it, you can download it from Figshare into the data subdirectory using:

```
dir.create("data", showWarnings = FALSE)
download.file(url = "https://ndownloader.figshare.com/files/2292171",
              destfile = "data/portal_mammals.sqlite", mode = "wb")
```

Connecting to databases

We can point R to this database using:

```
library(dplyr)
library(dbplyr)
#>
#> Attaching package: 'dbplyr'
#> The following objects are masked from 'package:dplyr':
#>
#>     ident, sql
mammals <- DBI::dbConnect(RSQLite::SQLite(), "data/portal_mammals.sqlite")
```

This command uses 2 packages that helps **dbplyr** and **dplyr** talk to the SQLite database. **DBI** is not something that you'll use directly as a user. It allows R to send commands to databases irrespective of the database management system used. The **RSQLite** package allows R to interface with SQLite databases.

This command does not load the data into the R session (as the `read_csv()` function did). Instead, it merely instructs R to connect to the SQLite database contained in the `portal_mammals.sqlite` file.

Using a similar approach, you could connect to many other database management systems that are supported by R including MySQL, PostgreSQL, BigQuery, etc.

Let's take a closer look at the mammals database we just connected to:

```
src_dbi(mammals)
```

```
#> src:  sqlite 3.19.3 [/home/travis/build/datacarpentry/R-ecology-
lesson/data/portal_mammals.sqlite]
#> tbls: plots, species, surveys
```

Just like a spreadsheet with multiple worksheets, a SQLite database can contain multiple tables. In this case three of them are listed in the tbls row in the output above:

- plots
- species
- surveys

Now that we know we can connect to the database, let's explore how to get the data from its tables into R.

Querying the database with the SQL syntax

To connect to tables within a database, you can use the tbl() function from **dplyr**. This function can be used to send SQL queries to the database. To demonstrate this functionality, let's select the columns "year", "species_id", and "plot_id" from the surveys table:

```
tbl(mammals, sql("SELECT year, species_id, plot_id FROM surveys"))
```

With this approach you can use any of the SQL queries we have seen in the database lesson.

Querying the database with the dplyr syntax

One of the strengths of **dplyr** is that the same operation can be done using **dplyr**'s verbs instead of writing SQL. First, we select the table on which to do the operations by creating the surveys object, and then we use the standard **dplyr** syntax as if it were a data frame:

```
surveys <- tbl(mammals, "surveys")
surveys %>%
    select(year, species_id, plot_id)
```

In this case, the surveys object behaves like a data frame. Several functions that can be used with data frames can also be used on tables from a database. For instance, the head() function can be used to check the first 10 rows of the table:

```
head(surveys, n = 10)
```

```
#> # Source:   lazy query [?? x 9]
#> # Database: sqlite 3.19.3
#> #   [/home/travis/build/datacarpentry/R-ecology-
lesson/data/portal_mammals.sqlite]
#>    record_id month   day  year plot_id species_id sex   hindfoot_length
#>        <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
#>  1         1     7    16  1977       2 NL         M                  32
#>  2         2     7    16  1977       3 NL         M                  33
#>  3         3     7    16  1977       2 DM         F                  37
#>  4         4     7    16  1977       7 DM         M                  36
#>  5         5     7    16  1977       3 DM         M                  35
#>  6         6     7    16  1977       1 PF         M                  14
#>  7         7     7    16  1977       2 PE         F                  NA
#>  8         8     7    16  1977       1 DM         M                  37
#>  9         9     7    16  1977       1 DM         F                  34
#> 10        10     7    16  1977       6 PF         F                  20
#> # ... with more rows, and 1 more variable: weight <int>
```

This output of the head command looks just like a regular `data.frame`: The table has 9 columns and the `head()` command shows us the first 10 rows. Note that the columns `plot_type`, `taxa`, `genus`, and `species`are missing. These are now located in the tables `plots` and `species` which we will join together in a moment.

However, some functions don't work quite as expected. For instance, let's check how many rows there are in total using `nrow()`:

```
nrow(tbl)
```

```
#> NULL
```

That's strange - R doesn't know how many rows the survey table contains - it returns NULL instead. You might have already noticed that the first line of the `head()` output included ?? indicating that the number of rows wasn't known.

The reason for this behavior highlights a key difference between using **`dplyr`** on datasets in memory (e.g. loaded into your R session via `read_csv()`) and those provided by a database. To understand it, we take a closer look at how **`dplyr`** communicates with our SQLite database.

SQL translation
Relational databases typically use a special-purpose language, Structured Query Language (SQL), to manage and query data.

For example, the following SQL query returns the first 10 rows from the surveys table:

```
SELECT *
FROM `surveys`
LIMIT 10
```

Behind the scenes, **dplyr**:

1. translates your R code into SQL
2. submits it to the database
3. translates the database's response into an R data frame

To lift the curtain, we can use **dplyr**'s `show_query()` function to show which SQL commands are actually sent to the database:

```
show_query(head(surveys, n = 10))
#> <SQL>
#> SELECT *
#> FROM `surveys`
#> LIMIT 10
```

The output shows the actual SQL query sent to the database; it matches our manually constructed `SELECT`statement above.

Instead of having to formulate the SQL query ourselves - and having to mentally switch back and forth between R and SQL syntax - we can delegate this translation to **dplyr**. (You don't even need to know SQL to interact with a database via **dplyr**!)

dplyr, in turn, doesn't do the real work of subsetting the table, either. Instead, it merely sends the query to the database, waits for its response and returns it to us.

That way, R never gets to see the full `surveys` table - and that's why it could not tell us how many rows it contains. On the bright side, this allows us to work with large datasets - even too large to fit into our computer's memory.

dplyr can translate many different query types into SQL allowing us to, e.g., `select()` specific columns, `filter()` rows, or join tables.

To see this in action, let's compose a few queries with **dplyr**.

Simple database queries

First, let's only request rows of the surveys table in which `weight` is less than 5 and keep only the species_id, sex, and weight columns.

```
surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)
```

```
#> # Source:   lazy query [?? x 3]
#> # Database: sqlite 3.19.3
#> #   [/home/travis/build/datacarpentry/R-ecology-
lesson/data/portal_mammals.sqlite]
#>    species_id sex   weight
#>    <chr>      <chr>  <int>
#>  1 PF         M          4
#>  2 PF         F          4
#>  3 PF         <NA>       4
#>  4 PF         F          4
#>  5 PF         F          4
#>  6 RM         M          4
#>  7 RM         F          4
#>  8 RM         M          4
#>  9 RM         M          4
#> 10 RM         M          4
#> # ... with more rows
```

Executing this command will return a table with 10 rows and the requested `species_id`, `sex` and `weight`columns. Great!

… but wait, why are there only 10 rows?

The last line:

```
# ... with more rows
```

indicates that there are more results that fit our filtering criterion. Why was R lazy and only retrieved 10 of them?

Laziness

Hadley Wickham, the author of **`dplyr`** explains:

When working with databases, dplyr tries to be as lazy as possible:

- It never pulls data into R unless you explicitly ask for it.

- It delays doing any work until the last possible moment - it collects together everything you want to do and then sends it to the database in one step.

When you construct a **dplyr** query, you can connect multiple verbs into a single pipeline. For example, we combined the `filter()` and `select()` verbs using the %>% pipe.

If we wanted to, we could add on even more steps, e.g. remove the `sex` column in an additional select call:

```
data_subset <- surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)


data_subset %>%
  select(-sex)
```

```
#> # Source:   lazy query [?? x 2]
#> # Database: sqlite 3.19.3
#> #   [/home/travis/build/datacarpentry/R-ecology-
lesson/data/portal_mammals.sqlite]
#>    species_id weight
#>    <chr>       <int>
#>  1 PF              4
#>  2 PF              4
#>  3 PF              4
#>  4 PF              4
#>  5 PF              4
#>  6 RM              4
#>  7 RM              4
#>  8 RM              4
#>  9 RM              4
#> 10 RM              4
#> # ... with more rows
```

Just like the first `select(species_id, sex, weight)` call, the `select(-sex)` command is not executed by R. It is sent to the database instead. Only the *final* result is retrieved and displayed to you.

Of course, we could always add on more steps, e.g., we could filter by `species_id` or minimum `weight`. That's why R doesn't retrieve the full set of results - instead it only

retrieves the first 10 results from the database by default. (After all, you might want to add an additional step and get the database to do more work…)

To instruct R to stop being lazy, e.g. to retrieve all of the query results from the database, we add the `collect()` command to our pipe. It indicates that our database query is finished: time to get the *final* results and load them into the R session.

```
data_subset <- surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight) %>%
  collect()
```

Now we have all 17 rows that match our query in a `data.frame` and can continue to work with them exclusively in R, without communicating with the database.

Complex database queries

`dplyr` enables database queries across one or multiple database tables, using the same single- and multiple-table verbs you encountered previously. This means you can use the same commands regardless of whether you interact with a remote database or local dataset! This is a really useful feature if you work with large datasets: you can first prototype your code on a small subset that fits into memory, and when your code is ready, you can change the input dataset to your full database without having to change the syntax.

On the other hand, being able use SQL queries directly can be useful if your collaborators have already put together complex queries to prepare the dataset that you need for your analysis.

To illustrate how to use **`dplyr`** with these complex queries, we are going to join the `plots` and `surveys`tables. The `plots` table in the database contains information about the different plots surveyed by the researchers. To access it, we point the `tbl()` command to it:

```
plots <- tbl(mammals, "plots")
plots
```

```
#> # Source:   table<plots> [?? x 2]
#> # Database: sqlite 3.19.3
#> #   [/home/travis/build/datacarpentry/R-ecology-
lesson/data/portal_mammals.sqlite]
#>    plot_id plot_type
#>      <int> <chr>
#>  1       1 Spectab exclosure
#>  2       2 Control
```

```
#>  3       3 Long-term Krat Exclosure
#>  4       4 Control
#>  5       5 Rodent Exclosure
#>  6       6 Short-term Krat Exclosure
#>  7       7 Rodent Exclosure
#>  8       8 Control
#>  9       9 Spectab exclosure
#> 10      10 Rodent Exclosure
#> # ... with more rows
```

The `plot_id` column also features in the `surveys` table:

```
surveys
#> # Source:   table<surveys> [?? x 9]
#> # Database: sqlite 3.19.3
#> #   [/home/travis/build/datacarpentry/R-ecology-
lesson/data/portal_mammals.sqlite]
#>    record_id month   day  year plot_id species_id sex
hindfoot_length
#>        <int> <int> <int> <int>   <int> <chr>      <chr>
<int>
#>  1         1     7    16  1977       2 NL         M
32
#>  2         2     7    16  1977       3 NL         M
33
#>  3         3     7    16  1977       2 DM         F
37
#>  4         4     7    16  1977       7 DM         M
36
#>  5         5     7    16  1977       3 DM         M
35
#>  6         6     7    16  1977       1 PF         M
14
#>  7         7     7    16  1977       2 PE         F
NA
#>  8         8     7    16  1977       1 DM         M
37
```

```
#>  9          9     7    16  1977       1 DM          F
34
#> 10         10     7    16  1977       6 PF          F
20
#> # ... with more rows, and 1 more variable: weight <int>
```

Because `plot_id` is listed in both tables, we can use it to look up matching records, and join the two tables.

diagram illustrating inner and left joins

For example, to extract all surveys for the first plot, which has `plot_id` 1, we can do:

```
plots %>%
  filter(plot_id == 1) %>%
  inner_join(surveys) %>%
  collect()
```

```
#> Joining, by = "plot_id"
#> # A tibble: 1,995 x 10
#>    plot_id plot_type         record_id month   day  year
species_id sex
#>      <int> <chr>                 <int> <int> <int> <int>
<chr>      <chr>
#>  1       1 Spectab exclosure         6     7    16  1977 PF
M
#>  2       1 Spectab exclosure         8     7    16  1977 DM
M
#>  3       1 Spectab exclosure         9     7    16  1977 DM
F
#>  4       1 Spectab exclosure        78     8    19  1977 PF
M
#>  5       1 Spectab exclosure        80     8    19  1977 DS
M
#>  6       1 Spectab exclosure       218     9    13  1977 PF
M
#>  7       1 Spectab exclosure       222     9    13  1977 DS
M
```

```
#>  8       1 Spectab exclosure       239     9    13  1977 DS
M
#>  9       1 Spectab exclosure       263    10    16  1977 DM
M
#> 10       1 Spectab exclosure       270    10    16  1977 DM
F
#> # ... with 1,985 more rows, and 2 more variables:
hindfoot_length <int>,
#> #   weight <int>
```

Important Note: Without the `collect()` statement, only the first 10 matching rows are returned. By adding `collect()`, the full set of 1,985 is retrieved.

Challenge

Write a query that returns the number of rodents observed in each plot in each year.

Hint: Connect to the species table and write a query that joins the species and survey tables together to exclude all non-rodents. The query should return counts of rodents by year.

Optional: Write a query in SQL that will produce the same result. You can join multiple tables together using the following syntax where foreign key refers to your unique id (e.g., `species_id`):

```
SELECT table.col, table.col
FROM table1 JOIN table2
ON table1.key = table2.key
JOIN table3 ON table2.key = table3.key
```

Challenge

Write a query that returns the total number of rodents in each genus caught in the different plot types.

Hint: Write a query that joins the species, plot, and survey tables together. The query should return counts of genus by plot type.

This is useful if we are interested in estimating the number of individuals belonging to each genus found in each plot type. But what if we were interested in the number of genera found in each plot type? Using `tally()` gives the number of individuals, instead we need to use `n_distinct()` to count the number of unique values found in a column.

```
unique_genera <- left_join(surveys, plots) %>%
    left_join(species) %>%
```

```
    group_by(plot_type) %>%
    summarize(
        n_genera = n_distinct(genus)
    ) %>%
    collect()
#> Joining, by = "plot_id"
#> Joining, by = "species_id"
```

`n_distinct`, like the other **`dplyr`** functions we have used in this lesson, works not only on database connections but also on regular data frames.

Creating a new SQLite database

So far, we have used a previously prepared SQLite database. But we can also use R to create a new database, e.g. from existing `csv` files. Let's recreate the mammals database that we've been working with, in R. First let's read in the `csv` files.

```
species <- read_csv("data/species.csv")
#> Parsed with column specification:
#> cols(
#>   species_id = col_character(),
#>   genus = col_character(),
#>   species = col_character(),
#>   taxa = col_character()
#> )
surveys <- read_csv("data/surveys.csv")
#> Parsed with column specification:
#> cols(
#>   record_id = col_integer(),
#>   month = col_integer(),
#>   day = col_integer(),
#>   year = col_integer(),
#>   plot_id = col_integer(),
#>   species_id = col_character(),
#>   sex = col_character(),
#>   hindfoot_length = col_integer(),
```

```
#>   weight = col_integer()
#> )
plots <- read_csv("data/plots.csv")
#> Parsed with column specification:
#> cols(
#>   plot_id = col_integer(),
#>   plot_type = col_character()
#> )
```

Creating a new SQLite database with **dplyr** is easy. You can re-use the same command we used above to open an existing `.sqlite` file. The `create = TRUE` argument instructs R to create a new, empty database instead.

Caution: When `create = TRUE` is added, any existing database at the same location is overwritten *without warning*.

```
my_db_file <- "portal-database.sqlite"
my_db <- src_sqlite(my_db_file, create = TRUE)
```

Currently, our new database is empty, it doesn't contain any tables:

```
my_db
#> src:  sqlite 3.19.3 [portal-database.sqlite]
#> tbls:
```

To add tables, we copy the existing data.frames into the database one by one:

```
copy_to(my_db, surveys)
copy_to(my_db, plots)
my_db
```

If you check the location of our database you'll see that data is automatically being written to disk. R and **dplyr** not only provide easy ways to query existing databases, they also allows you to easily create your own databases from flat files!

Challenge

Add the remaining species table to the `my_db` database and run some of your queries from earlier in the lesson to verify that you have faithfully recreated the mammals database.

Note: In this example, we first loaded all of the data into the R session by reading the three `csv` files. Because all the data has to flow through R, this is not suitable for very large datasets.

It's rare that a data analysis involves only a single table of data. Typically you have many tables of data, and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called **relational data** because it is the relations, not just the individual datasets, that are important.

Relations are always defined between a pair of tables. All other relations are built up from this simple idea: the relations of three or more tables are always a property of the relations between each pair. Sometimes both elements of a pair can be the same table! This is needed if, for example, you have a table of people, and each person has a reference to their parents.

To work with relational data you need verbs that work with pairs of tables. There are three families of verbs designed to work with relational data:

- **Mutating joins**, which add new variables to one data frame from matching observations in another.
- **Filtering joins**, which filter observations from one data frame based on whether or not they match an observation in the other table.
- **Set operations**, which treat observations as if they were set elements.

The most common place to find relational data is in a *relational* database management system (or RDBMS), a term that encompasses almost all modern databases. If you've used a database before, you've almost certainly used SQL. If so, you should find the concepts in this chapter familiar, although their expression in dplyr is a little different. Generally, dplyr is a little easier to use than SQL because dplyr is specialised to do data analysis: it makes common data analysis operations easier, at the expense of making it more difficult to do other things that aren't commonly needed for data analysis.

13.1.1 Prerequisites

We will explore relational data from `nycflights13` using the two-table verbs from dplyr.

```
library(tidyverse)
library(nycflights13)
```

13.2 nycflights13

We will use the nycflights13 package to learn about relational data. nycflights13 contains four tibbles that are related to the `flights` table that you used in data transformation:

- `airlines` lets you look up the full carrier name from its abbreviated code:

```
airlines
#> # A tibble: 16 × 2
#>   carrier                    name
#>     <chr>                   <chr>
```

```
#> 1      9E        Endeavor Air Inc.
#> 2      AA   American Airlines Inc.
#> 3      AS     Alaska Airlines Inc.
#> 4      B6          JetBlue Airways
#> 5      DL     Delta Air Lines Inc.
#> 6      EV ExpressJet Airlines Inc.
#> # ... with 10 more rows
```

`airports` gives information about each airport, identified by the `faa` airport code:

```
airports
#> # A tibble: 1,458 × 8
#>    faa                           name   lat   lon   alt    tz
dst
#>   <chr>                         <chr> <dbl> <dbl> <int> <dbl>
<chr>
#> 1   04G              Lansdowne Airport  41.1 -80.6  1044    -5
A
#> 2   06A  Moton Field Municipal Airport  32.5 -85.7   264    -6
A
#> 3   06C            Schaumburg Regional  42.0 -88.1   801    -6
A
#> 4   06N                Randall Airport  41.4 -74.4   523    -5
A
#> 5   09J          Jekyll Island Airport  31.1 -81.4    11    -5
A
#> 6   0A9 Elizabethton Municipal Airport  36.4 -82.2  1593    -5
A
#> # ... with 1,452 more rows, and 1 more variables: tzone <chr>
```

`planes` gives information about each plane, identified by its `tailnum`:

```
planes
#> # A tibble: 3,322 × 9
#>   tailnum  year                type     manufacturer     model
engines
#>     <chr> <int>                <chr>            <chr>     <chr>
<int>
#> 1  N10156  2004 Fixed wing multi engine          EMBRAER EMB-145XR
2
#> 2  N102UW  1998 Fixed wing multi engine AIRBUS INDUSTRIE  A320-214
2
#> 3  N103US  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A320-214
2
#> 4  N104UW  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A320-214
2
#> 5  N10575  2002 Fixed wing multi engine          EMBRAER EMB-145LR
2
#> 6  N105UW  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A320-214
2
```

- `#> # ... with 3,316 more rows, and 3 more variables: seats <int>, #> #   speed <int>, engine <chr>`
- `weather` gives the weather at each NYC airport for each hour:
- `weather`
- `#> # A tibble: 26,130 × 15`
- `#>   origin  year month   day  hour  temp  dewp humid wind_dir wind_speed`
- `#>    <chr> <dbl> <dbl> <int> <int> <dbl> <dbl> <dbl>    <dbl>      <dbl>`
- `#> 1    EWR  2013     1     1     0  37.0  21.9  54.0      230       10.4`
- `#> 2    EWR  2013     1     1     1  37.0  21.9  54.0      230       13.8`
- `#> 3    EWR  2013     1     1     2  37.9  21.9  52.1      230       12.7`
- `#> 4    EWR  2013     1     1     3  37.9  23.0  54.5      230       13.8`
- `#> 5    EWR  2013     1     1     4  37.9  24.1  57.0      240       15.0`
- `#> 6    EWR  2013     1     1     6  39.0  26.1  59.4      270       10.4`
- `#> # ... with 2.612e+04 more rows, and 5 more variables: wind_gust <dbl>, #> #   precip <dbl>, pressure <dbl>, visib <dbl>, time_hour <dttm>`

One way to show the relationships between the different tables is with a drawing:

This diagram is a little overwhelming, but it's simple compared to some you'll see in the wild! The key to understanding diagrams like this is to remember each relation always concerns a pair of tables. You don't need to understand the whole thing; you just need to understand the chain of relations between the tables that you are interested in.

For nycflights13:

- `flights` connects to `planes` via a single variable, `tailnum`.
- `flights` connects to `airlines` through the `carrier` variable.
- `flights` connects to `airports` in two ways: via the `origin` and `dest` variables.
- `flights` connects to `weather` via `origin` (the location), and `year`, `month`, `day` and `hour` (the time).

13.3 Keys

The variables used to connect each pair of tables are called **keys**. A key is a variable (or set of variables) that uniquely identifies an observation. In simple cases, a single variable is sufficient to identify an observation. For example, each plane is uniquely identified by its `tailnum`. In other cases, multiple variables may be needed. For example, to identify an observation in `weather` you need five variables: `year`, `month`, `day`, `hour`, and `origin`.
There are two types of keys:

- A **primary key** uniquely identifies an observation in its own table. For example, `planes$tailnum` is a primary key because it uniquely identifies each plane in the `planes` table.
- A **foreign key** uniquely identifies an observation in another table. For example, the `flights$tailnum`is a foreign key because it appears in the `flights` table where it matches each flight to a unique plane.

A variable can be both a primary key *and* a foreign key. For example, `origin` is part of the `weather`primary key, and is also a foreign key for the `airport` table.
Once you've identified the primary keys in your tables, it's good practice to verify that they do indeed uniquely identify each observation. One way to do that is to `count()` the primary keys and look for entries where `n` is greater than one:

```
planes %>%
  count(tailnum) %>%
  filter(n > 1)
#> # A tibble: 0 × 2
#> # ... with 2 variables: tailnum <chr>, n <int>

weather %>%
  count(year, month, day, hour, origin) %>%
  filter(n > 1)
#> Source: local data frame [0 x 6]
#> Groups: year, month, day, hour [0]
#>
#> # ... with 6 variables: year <dbl>, month <dbl>, day <int>, hour
<int>,
#> #   origin <chr>, n <int>
```

Sometimes a table doesn't have an explicit primary key: each row is an observation, but no combination of variables reliably identifies it. For example, what's the primary key in the `flights` table? You might think it would be the date plus the flight or tail number, but neither of those are unique:

```
flights %>%
  count(year, month, day, flight) %>%
  filter(n > 1)
#> Source: local data frame [29,768 x 5]
#> Groups: year, month, day [365]
#>
#>    year month   day flight     n
#>   <int> <int> <int>  <int> <int>
```

```
#> 1  2013     1     1      1     2
#> 2  2013     1     1      3     2
#> 3  2013     1     1      4     2
#> 4  2013     1     1     11     3
#> 5  2013     1     1     15     2
#> 6  2013     1     1     21     2
#> # ... with 2.976e+04 more rows

flights %>%
  count(year, month, day, tailnum) %>%
  filter(n > 1)
#> Source: local data frame [64,928 x 5]
#> Groups: year, month, day [365]
#>
#>    year month   day tailnum     n
#>   <int> <int> <int>   <chr> <int>
#> 1  2013     1     1  N0EGMQ     2
#> 2  2013     1     1  N11189     2
#> 3  2013     1     1  N11536     2
#> 4  2013     1     1  N11544     3
#> 5  2013     1     1  N11551     2
#> 6  2013     1     1  N12540     2
#> # ... with 6.492e+04 more rows
```

When starting to work with this data, I had naively assumed that each flight number would be only used once per day: that would make it much easier to communicate problems with a specific flight. Unfortunately that is not the case! If a table lacks a primary key, it's sometimes useful to add one with `mutate()` and `row_number()`. That makes it easier to match observations if you've done some filtering and want to check back in with the original data. This is called a **surrogate key**.

A primary key and the corresponding foreign key in another table form a **relation**. Relations are typically one-to-many. For example, each flight has one plane, but each plane has many flights. In other data, you'll occasionally see a 1-to-1 relationship. You can think of this as a special case of 1-to-many. You can model many-to-many relations with a many-to-1 relation plus a 1-to-many relation. For example, in this data there's a many-to-many relationship between airlines and airports: each airline flies to many airports; each airport hosts many airlines.

Exercises

1. Add a surrogate key to `flights`.
2. Identify the keys in the following datasets

1. `Lahman::Batting`,
2. `babynames::babynames`
3. `nasaweather::atmos`
4. `fueleconomy::vehicles`

5. `ggplot2::diamonds`

   (You might need to install some packages and read some documentation.)

3. Draw a diagram illustrating the connections between the `Batting`, `Master`, and `Salaries` tables in the Lahman package. Draw another diagram that shows the relationship between `Master`, `Managers`, `AwardsManagers`.
   How would you characterise the relationship between the `Batting`, `Pitching`, and `Fieldingtables`?

13.4 Mutating joins

The first tool we'll look at for combining a pair of tables is the **mutating join**. A mutating join allows you to combine variables from two tables. It first matches observations by their keys, then copies across variables from one table to the other.

Like `mutate()`, the join functions add variables to the right, so if you have a lot of variables already, the new variables won't get printed out. For these examples, we'll make it easier to see what's going on in the examples by creating a narrower dataset:

```
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
flights2
#> # A tibble: 336,776 × 8
#>    year month   day  hour origin  dest tailnum carrier
#>   <int> <int> <int> <dbl>  <chr> <chr>   <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228      UA
#> 2  2013     1     1     5    LGA   IAH  N24211      UA
#> 3  2013     1     1     5    JFK   MIA  N619AA      AA
#> 4  2013     1     1     5    JFK   BQN  N804JB      B6
#> 5  2013     1     1     6    LGA   ATL  N668DN      DL
#> 6  2013     1     1     5    EWR   ORD  N39463      UA
#> # ... with 3.368e+05 more rows
```

(Remember, when you're in RStudio, you can also use `View()` to avoid this problem.)
Imagine you want to add the full airline name to the `flights2` data. You can combine the `airlines` and `flights2` data frames with `left_join()`:

```
flights2 %>%
  select(-origin, -dest) %>%
  left_join(airlines, by = "carrier")
#> # A tibble: 336,776 × 7
#>    year month   day  hour tailnum carrier                    name
#>   <int> <int> <int> <dbl>   <chr>   <chr>                   <chr>
#> 1  2013     1     1     5  N14228      UA  United Air Lines Inc.
#> 2  2013     1     1     5  N24211      UA  United Air Lines Inc.
#> 3  2013     1     1     5  N619AA      AA American Airlines Inc.
#> 4  2013     1     1     5  N804JB      B6        JetBlue Airways
#> 5  2013     1     1     6  N668DN      DL   Delta Air Lines Inc.
```

```
#> 6  2013     1     1     5  N39463      UA  United Air Lines Inc.
#> # ... with 3.368e+05 more rows
```

The result of joining airlines to flights2 is an additional variable: `name`. This is why I call this type of join a mutating join. In this case, you could have got to the same place using `mutate()` and R's base subsetting:

```
flights2 %>%
  select(-origin, -dest) %>%
  mutate(name = airlines$name[match(carrier, airlines$carrier)])
#> # A tibble: 336,776 × 7
#>    year month   day  hour tailnum carrier                  name
#>   <int> <int> <int> <dbl>   <chr>   <chr>                 <chr>
#> 1  2013     1     1     5  N14228      UA  United Air Lines Inc.
#> 2  2013     1     1     5  N24211      UA  United Air Lines Inc.
#> 3  2013     1     1     5  N619AA      AA American Airlines Inc.
#> 4  2013     1     1     5  N804JB      B6        JetBlue Airways
#> 5  2013     1     1     6  N668DN      DL   Delta Air Lines Inc.
#> 6  2013     1     1     5  N39463      UA  United Air Lines Inc.
#> # ... with 3.368e+05 more rows
```

But this is hard to generalise when you need to match multiple variables, and takes close reading to figure out the overall intent.

The following sections explain, in detail, how mutating joins work. You'll start by learning a useful visual representation of joins. We'll then use that to explain the four mutating join functions: the inner join, and the three outer joins. When working with real data, keys don't always uniquely identify observations, so next we'll talk about what happens when there isn't a unique match. Finally, you'll learn how to tell dplyr which variables are the keys for a given join.

13.4.1 Understanding joins

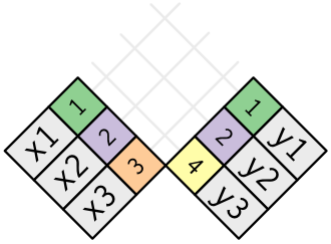
To help you learn how joins work, I'm going to use a visual representation:

```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
```

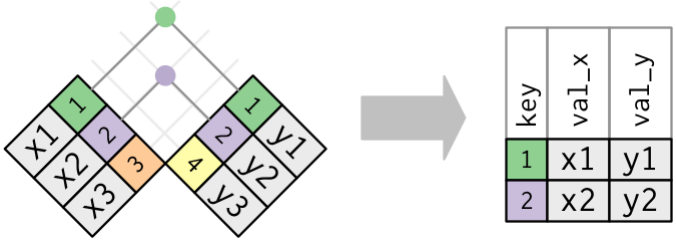
```
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```

The coloured column represents the “key” variable: these are used to match the rows between the tables. The grey column represents the “value” column that is carried along for the ride. In these examples I’ll show a single key variable, but the idea generalises in a straightforward way to multiple keys and multiple values.

A join is a way of connecting each row in x to zero, one, or more rows in y. The following diagram shows each potential match as an intersection of a pair of lines.

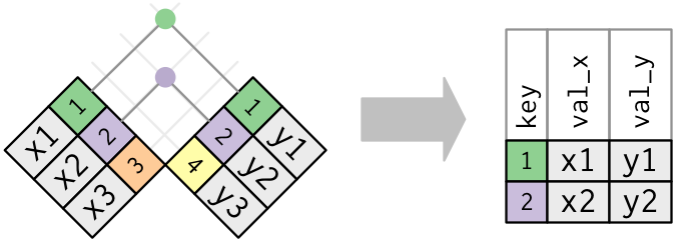


(If you look closely, you might notice that we’ve switched the order of the key and value columns in x. This is to emphasise that joins match based on the key; the value is just carried along for the ride.)
In an actual join, matches will be indicated with dots. The number of dots = the number of matches = the number of rows in the output.



13.4.2 Inner join

The simplest type of join is the **inner join**. An inner join matches pairs of observations whenever their keys are equal:



(To be precise, this is an inner **equijoin** because the keys are matched using the equality operator. Since most joins are equijoins we usually drop that specification.)

The output of an inner join is a new data frame that contains the key, the x values, and the y values. We use by to tell dplyr which variable is the key:

```
x %>%
  inner_join(y, by = "key")
#> # A tibble: 2 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1    x1    y1
#> 2     2    x2    y2
```

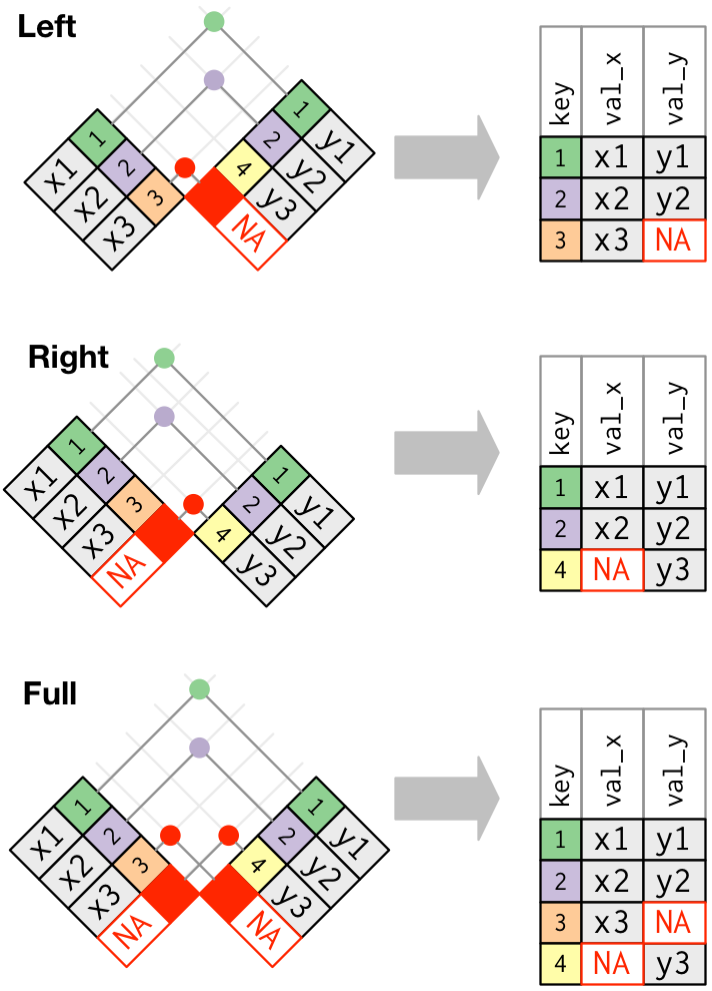
The most important property of an inner join is that unmatched rows are not included in the result. This means that generally inner joins are usually not appropriate for use in analysis because it's too easy to lose observations.

13.4.3 Outer joins

An inner join keeps observations that appear in both tables. An **outer join** keeps observations that appear in at least one of the tables. There are three types of outer joins:

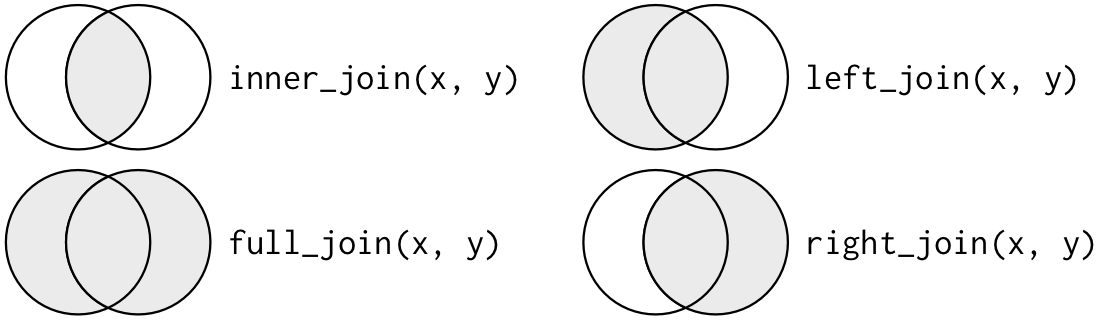
- A **left join** keeps all observations in x.
- A **right join** keeps all observations in y.
- A **full join** keeps all observations in x and y.

These joins work by adding an additional "virtual" observation to each table. This observation has a key that always matches (if no other key matches), and a value filled with NA.
Graphically, that looks like:



The most commonly used join is the left join: you use this whenever you look up additional data from another table, because it preserves the original observations even when there isn't a match. The left join should be your default join: use it unless you have a strong reason to prefer one of the others.

Another way to depict the different types of joins is with a Venn diagram:



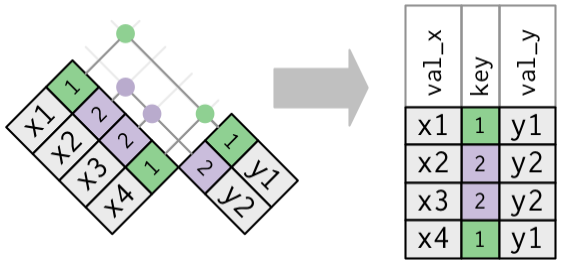
However, this is not a great representation. It might jog your memory about which join preserves the observations in which table, but it suffers from a major limitation: a

Venn diagram can't show what happens when keys don't uniquely identify an observation.

13.4.4 Duplicate keys

So far all the diagrams have assumed that the keys are unique. But that's not always the case. This section explains what happens when the keys are not unique. There are two possibilities:

1. One table has duplicate keys. This is useful when you want to add in additional information as there is typically a one-to-many relationship.

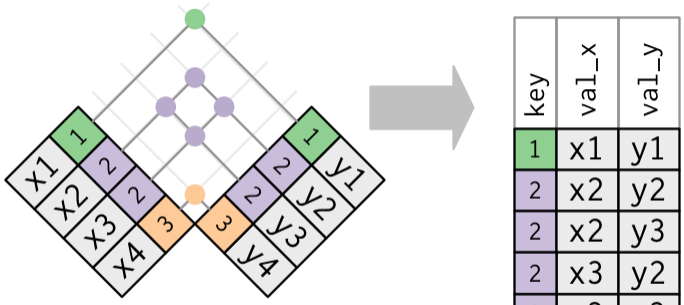


val_x	key	val_y
x1	1	y1
x2	2	y2
x3	2	y2
x4	1	y1

Note that I've put the key column in a slightly different position in the output. This reflects that the key is a primary key in y and a foreign key in x.

```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     2, "x3",
     1, "x4"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2"
)
left_join(x, y, by = "key")
#> # A tibble: 4 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1    x1    y1
#> 2     2    x2    y2
#> 3     2    x3    y2
#> 4     1    x4    y1
```

2. Both tables have duplicate keys. This is usually an error because in neither table do the keys uniquely identify an observation. When you join duplicated keys, you get all possible combinations, the Cartesian product:



key	val_x	val_y
1	x1	y1
2	x2	y2
2	x2	y3
2	x3	y2
2	x3	y3
3	x4	y4

```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     2, "x3",
     3, "x4"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     2, "y3",
     3, "y4"
)
left_join(x, y, by = "key")
#> # A tibble: 6 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1    x1    y1
#> 2     2    x2    y2
#> 3     2    x2    y3
#> 4     2    x3    y2
#> 5     2    x3    y3
#> 6     3    x4    y4
```

13.4.5 Defining the key columns

So far, the pairs of tables have always been joined by a single variable, and that variable has the same name in both tables. That constraint was encoded by `by = "key"`. You can use other values for `by` to connect the tables in other ways:

- The default, `by = NULL`, uses all variables that appear in both tables, the so called **natural** join. For example, the flights and weather tables match on their common variables: `year`, `month`, `day`, `hour` and `origin`.

```
flights2 %>%
  left_join(weather)
```

```
#> Joining, by = c("year", "month", "day", "hour", "origin")
#> # A tibble: 336,776 × 18
#>    year month   day  hour origin  dest tailnum carrier  temp  dewp
humid
#>   <dbl> <dbl> <int> <dbl>  <chr> <chr>   <chr>   <chr> <dbl> <dbl>
<dbl>
#> 1  2013     1     1     5    EWR   IAH  N14228      UA    NA    NA
NA
#> 2  2013     1     1     5    LGA   IAH  N24211      UA    NA    NA
NA
#> 3  2013     1     1     5    JFK   MIA  N619AA      AA    NA    NA
NA
#> 4  2013     1     1     5    JFK   BQN  N804JB      B6    NA    NA
NA
#> 5  2013     1     1     6    LGA   ATL  N668DN      DL  39.9  26.1
57.3
#> 6  2013     1     1     5    EWR   ORD  N39463      UA    NA    NA
NA
#> # ... with 3.368e+05 more rows, and 7 more variables: wind_dir
<dbl>,
#> #   wind_speed <dbl>, wind_gust <dbl>, precip <dbl>, pressure
<dbl>,
#> #   visib <dbl>, time_hour <dttm>
```

- A character vector, `by = "x"`. This is like a natural join, but uses only some of the common variables. For example, `flights` and `planes` have `year` variables, but they mean different things so we only want to join by `tailnum`.

```
flights2 %>%
  left_join(planes, by = "tailnum")
#> # A tibble: 336,776 × 16
#>   year.x month   day  hour origin  dest tailnum carrier year.y
#>    <int> <int> <int> <dbl>  <chr> <chr>   <chr>   <chr>  <int>
#> 1   2013     1     1     5    EWR   IAH  N14228      UA   1999
#> 2   2013     1     1     5    LGA   IAH  N24211      UA   1998
#> 3   2013     1     1     5    JFK   MIA  N619AA      AA   1990
#> 4   2013     1     1     5    JFK   BQN  N804JB      B6   2012
#> 5   2013     1     1     6    LGA   ATL  N668DN      DL   1991
#> 6   2013     1     1     5    EWR   ORD  N39463      UA   2012
#> # ... with 3.368e+05 more rows, and 7 more variables: type <chr>,
#> #   manufacturer <chr>, model <chr>, engines <int>, seats <int>,
#> #   speed <int>, engine <chr>
```

  Note that the `year` variables (which appear in both input data frames, but are not constrained to be equal) are disambiguated in the output with a suffix.
- A named character vector: `by = c("a" = "b")`. This will match variable `a` in table `x` to variable `b` in table `y`. The variables from `x` will be used in the output.
  For example, if we want to draw a map we need to combine the flights data with the airports data which contains the location (`lat` and `long`) of each airport. Each flight

has an origin and destination `airport`, so we need to specify which one we want to join to:

```
flights2 %>%
  left_join(airports, c("dest" = "faa"))
#> # A tibble: 336,776 × 15
#>    year month   day  hour origin  dest tailnum carrier
#>   <int> <int> <int> <dbl>  <chr> <chr>   <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228      UA
#> 2  2013     1     1     5    LGA   IAH  N24211      UA
#> 3  2013     1     1     5    JFK   MIA  N619AA      AA
#> 4  2013     1     1     5    JFK   BQN  N804JB      B6
#> 5  2013     1     1     6    LGA   ATL  N668DN      DL
#> 6  2013     1     1     5    EWR   ORD  N39463      UA
#> # ... with 3.368e+05 more rows, and 7 more variables: name <chr>,
#> #   lat <dbl>, lon <dbl>, alt <int>, tz <dbl>, dst <chr>, tzone
<chr>

flights2 %>%
  left_join(airports, c("origin" = "faa"))
#> # A tibble: 336,776 × 15
#>    year month   day  hour origin  dest tailnum carrier
name
#>   <int> <int> <int> <dbl>  <chr> <chr>   <chr>   <chr>
<chr>
#> 1  2013     1     1     5    EWR   IAH  N14228      UA Newark
Liberty Intl
#> 2  2013     1     1     5    LGA   IAH  N24211      UA          La
Guardia
#> 3  2013     1     1     5    JFK   MIA  N619AA      AA John F
Kennedy Intl
#> 4  2013     1     1     5    JFK   BQN  N804JB      B6 John F
Kennedy Intl
#> 5  2013     1     1     6    LGA   ATL  N668DN      DL          La
Guardia
#> 6  2013     1     1     5    EWR   ORD  N39463      UA Newark
Liberty Intl
#> # ... with 3.368e+05 more rows, and 6 more variables: lat <dbl>,
#> #   lon <dbl>, alt <int>, tz <dbl>, dst <chr>, tzone <chr>
```

13.4.6 Exercises

1. Compute the average delay by destination, then join on the `airports` data frame so you can show the spatial distribution of delays. Here's an easy way to draw a map of the United States:

```
2. airports %>%
3.   semi_join(flights, c("faa" = "dest")) %>%
```

```
  ggplot(aes(lon, lat)) +
    borders("state") +
    geom_point() +
    coord_quickmap()
```

(Don't worry if you don't understand what `semi_join()` does — you'll learn about it next.)
You might want to use the `size` or `colour` of the points to display the average delay for each airport.

7. Add the location of the origin *and* destination (i.e. the `lat` and `lon`) to `flights`.
8. Is there a relationship between the age of a plane and its delays?

9. What weather conditions make it more likely to see a delay?

10. What happened on June 13 2013? Display the spatial pattern of delays, and then use Google to cross-reference with the weather.

13.4.7 Other implementations

`base::merge()` can perform all four types of mutating join:

dplyr	merge
`inner_join(x, y)`	`merge(x, y)`
`left_join(x, y)`	`merge(x, y, all.x = TRUE)`
`right_join(x, y)`	`merge(x, y, all.y = TRUE),`
`full_join(x, y)`	`merge(x, y, all.x = TRUE, all.y = TRUE)`

The advantages of the specific dplyr verbs is that they more clearly convey the intent of your code: the difference between the joins is really important but concealed in the arguments of `merge()`. dplyr's joins are considerably faster and don't mess with the order of the rows.
SQL is the inspiration for dplyr's conventions, so the translation is straightforward:

dplyr	SQL
`inner_join(x, y, by = "z")`	`SELECT * FROM x INNER JOIN y USING (z)`
`left_join(x, y, by = "z")`	`SELECT * FROM x LEFT OUTER JOIN y USING (z)`
`right_join(x, y, by = "z")`	`SELECT * FROM x RIGHT OUTER JOIN y USING (z)`
`full_join(x, y, by = "z")`	`SELECT * FROM x FULL OUTER JOIN y USING (z)`

Note that "INNER" and "OUTER" are optional, and often omitted.

Joining different variables between the tables, e.g. `inner_join(x, y, by = c("a" = "b"))` uses a slightly different syntax in SQL: `SELECT * FROM x INNER JOIN y ON x.a = y.b`. As this syntax suggests, SQL supports a wider range of join types

than dplyr because you can connect the tables using constraints other than equality (sometimes called non-equijoins).

13.5 Filtering joins

Filtering joins match observations in the same way as mutating joins, but affect the observations, not the variables. There are two types:

- `semi_join(x, y)` **keeps** all observations in `x` that have a match in `y`.
- `anti_join(x, y)` **drops** all observations in `x` that have a match in `y`.

Semi-joins are useful for matching filtered summary tables back to the original rows. For example, imagine you've found the top ten most popular destinations:

```
top_dest <- flights %>%
  count(dest, sort = TRUE) %>%
  head(10)
top_dest
#> # A tibble: 10 × 2
#>    dest     n
#>   <chr> <int>
#> 1   ORD 17283
#> 2   ATL 17215
#> 3   LAX 16174
#> 4   BOS 15508
#> 5   MCO 14082
#> 6   CLT 14064
#> # ... with 4 more rows
```

Now you want to find each flight that went to one of those destinations. You could construct a filter yourself:

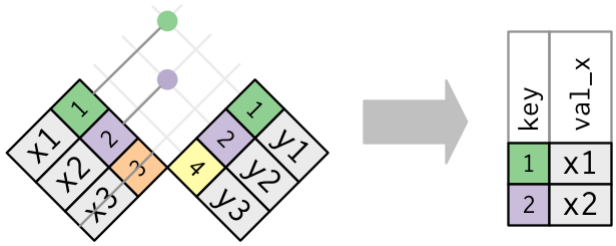
```
flights %>%
  filter(dest %in% top_dest$dest)
#> # A tibble: 141,145 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      542            540         2      923
#> 2  2013     1     1      554            600        -6      812
#> 3  2013     1     1      554            558        -4      740
#> 4  2013     1     1      555            600        -5      913
#> 5  2013     1     1      557            600        -3      838
#> 6  2013     1     1      558            600        -2      753
#> # ... with 1.411e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight
<int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
```

```
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

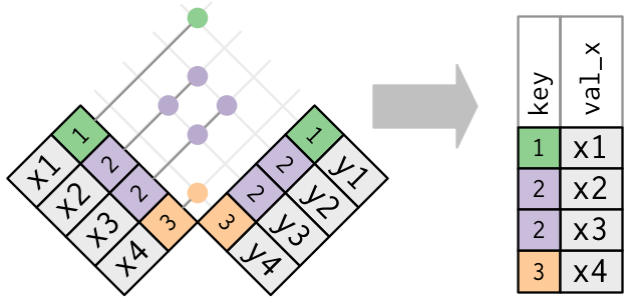
But it's difficult to extend that approach to multiple variables. For example, imagine that you'd found the 10 days with highest average delays. How would you construct the filter statement that used `year`, `month`, and `day` to match it back to `flights`? Instead you can use a semi-join, which connects the two tables like a mutating join, but instead of adding new columns, only keeps the rows in `x` that have a match in `y`:

```
flights %>%
  semi_join(top_dest)
#> Joining, by = "dest"
#> # A tibble: 141,145 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1  2013     1     1      554            558        -4      740
#> 2  2013     1     1      558            600        -2      753
#> 3  2013     1     1      608            600         8      807
#> 4  2013     1     1      629            630        -1      824
#> 5  2013     1     1      656            700        -4      854
#> 6  2013     1     1      709            700         9      852
#> # ... with 1.411e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight
<int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

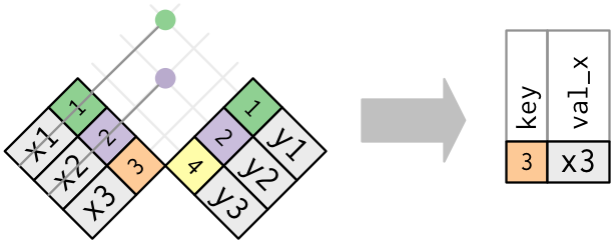
Graphically, a semi-join looks like this:



Only the existence of a match is important; it doesn't matter which observation is matched. This means that filtering joins never duplicate rows like mutating joins do:



The inverse of a semi-join is an anti-join. An anti-join keeps the rows that *don't* have a match:



Anti-joins are useful for diagnosing join mismatches. For example, when connecting `flights` and `planes`, you might be interested to know that there are many `flights` that don't have a match in `planes`:

```
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)
#> # A tibble: 722 × 2
#>   tailnum     n
#>     <chr> <int>
#> 1    <NA>  2512
#> 2  N725MQ   575
#> 3  N722MQ   513
#> 4  N723MQ   507
#> 5  N713MQ   483
#> 6  N735MQ   396
#> # ... with 716 more rows
```

13.5.1 Exercises

1. What does it mean for a flight to have a missing `tailnum`? What do the tail numbers that don't have a matching record in `planes` have in common? (Hint: one variable explains ~90% of the problems.)
2. Filter flights to only show flights with planes that have flown at least 100 flights.

3. Combine `fueleconomy::vehicles` and `fueleconomy::common` to find only the records for the most common models.
4. Find the 48 hours (over the course of the whole year) that have the worst delays. Cross-reference it with the `weather` data. Can you see any patterns?
5. What does `anti_join(flights, airports, by = c("dest" = "faa"))` tell you? What does `anti_join(airports, flights, by = c("faa" = "dest"))` tell you?
6. You might expect that there's an implicit relationship between plane and airline, because each plane is flown by a single airline. Confirm or reject this hypothesis using the tools you've learned above.

13.6 Join problems

The data you've been working with in this chapter has been cleaned up so that you'll have as few problems as possible. Your own data is unlikely to be so nice, so there

are a few things that you should do with your own data to make your joins go smoothly.

1. Start by identifying the variables that form the primary key in each table. You should usually do this based on your understanding of the data, not empirically by looking for a combination of variables that give a unique identifier. If you just look for variables without thinking about what they mean, you might get (un)lucky and find a combination that's unique in your current data but the relationship might not be true in general.

   For example, the altitude and longitude uniquely identify each airport, but they are not good identifiers!

   ```
   airports %>% count(alt, lon) %>% filter(n > 1)
   #> Source: local data frame [0 x 3]
   #> Groups: alt [0]
   #>
   #> # ... with 3 variables: alt <int>, lon <dbl>, n <int>
   ```

2. Check that none of the variables in the primary key are missing. If a value is missing then it can't identify an observation!

3. Check that your foreign keys match primary keys in another table. The best way to do this is with an `anti_join()`. It's common for keys not to match because of data entry errors. Fixing these is often a lot of work.
   If you do have missing keys, you'll need to be thoughtful about your use of inner vs. outer joins, carefully considering whether or not you want to drop rows that don't have a match.

   Be aware that simply checking the number of rows before and after the join is not sufficient to ensure that your join has gone smoothly. If you have an inner join with duplicate keys in both tables, you might get unlucky as the number of dropped rows might exactly equal the number of duplicated rows!

13.7 Set operations

The final type of two-table verb are the set operations. Generally, I use these the least frequently, but they are occasionally useful when you want to break a single complex filter into simpler pieces. All these operations work with a complete row, comparing the values of every variable. These expect the `x` and `y` inputs to have the same variables, and treat the observations like sets:

- `intersect(x, y)`: return only observations in both `x` and `y`.
- `union(x, y)`: return unique observations in `x` and `y`.
- `setdiff(x, y)`: return observations in `x`, but not in `y`.

Given this simple data:

```
df1 <- tribble(
```

```
  ~x, ~y,
   1,  1,
   2,  1
)
df2 <- tribble(
  ~x, ~y,
   1,  1,
   1,  2
)
```

The four possibilities are:

```
intersect(df1, df2)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     1

# Note that we get 3 rows, not 4
union(df1, df2)
#> # A tibble: 3 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
#> 2     2     1
#> 3     1     1

setdiff(df1, df2)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     2     1

setdiff(df2, df1)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
```

R-Programming

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many Robjects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validity	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name , var.name	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.
_var_name	invalid	Starts with _ which is not valid

Variable Assignment

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using **print()** or **cat()**function. The **cat()** function combines multiple items into a continuous print output.

```
# Assignment using equal operator.
var.1 = c(0,1,2,3)


# Assignment using leftward operator.
var.2 <- c("learn","R")


# Assignment using rightward operator.
```

```
c(TRUE,1) -> var.3


print(var.1)

cat ("var.1 is ", var.1 ,"\n")

cat ("var.2 is ", var.2 ,"\n")

cat ("var.3 is ", var.3 ,"\n")
```

When we execute the above code, it produces the following result –

```
[1] 0 1 2 3
var.1 is  0 1 2 3
var.2 is  learn R
var.3 is  1 1
```

Note – The vector c(TRUE,1) has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

Data Type of a Variable

In R, a variable itself is not declared of any data type, rather it gets the data type of the R - object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
var_x <- "Hello"

cat("The class of var_x is ",class(var_x),"\n")


var_x <- 34.5

cat("  Now the class of var_x is ",class(var_x),"\n")


var_x <- 27L

cat("   Next the class of var_x becomes ",class(var_x),"\n")
```

When we execute the above code, it produces the following result –

```
The class of var_x is  character
   Now the class of var_x is  numeric
      Next the class of var_x becomes  integer
```

Finding Variables

To know all the variables currently available in the workspace we use the **ls()**function. Also the ls() function can use patterns to match the variable names.

```
print(ls())
```

When we execute the above code, it produces the following result –

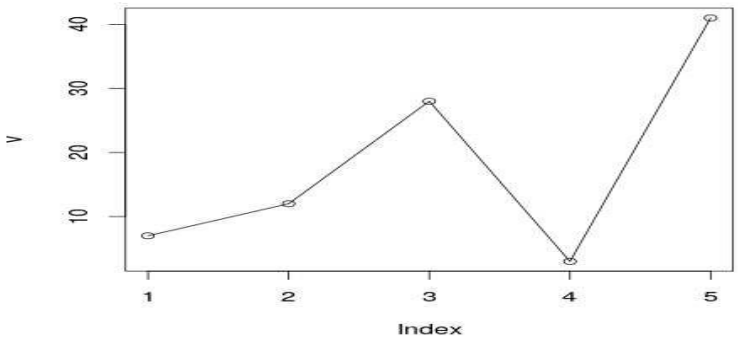
```
[1] "my var"     "my_new_var" "my_var"     "var.1"
[5] "var.2"      "var.3"      "var.name"   "var_name2."
[9] "var_x"      "varname"
```

Note – It is a sample output depending on what variables are declared in your environment.

The ls() function can use patterns to match the variable names.

 # List the variables starting with the pattern "var".
print(ls(pattern = "var"))

Executing the program....
$Rscript main.r
```
character(0)
```



When we execute the above code, it produces the following result –

```
[1] "my var"     "my_new_var" "my_var"     "var.1"
[5] "var.2"      "var.3"      "var.name"   "var_name2."
[9] "var_x"      "varname"
```

Note – It is a sample output depending on what variables are declared in your environment.

The ls() function can use patterns to match the variable names.

```
# List the variables starting with the pattern "var".
```

```
print(ls(pattern = "var"))
```

When we execute the above code, it produces the following result –

```
[1] "my var"     "my_new_var" "my_var"     "var.1"
[5] "var.2"      "var.3"      "var.name"   "var_name2."
[9] "var_x"      "varname"
```

The variables starting with **dot(.)** are hidden, they can be listed using "all.names = TRUE" argument to ls() function.

```
print(ls(all.name = TRUE))
```

When we execute the above code, it produces the following result –

```
[1] ".cars"        ".Random.seed" ".var_name"    ".varname"
".varname2"
[6] "my var"       "my_new_var"   "my_var"       "var.1"
"var.2"
[11]"var.3"        "var.name"     "var_name2."   "var_x"
```

Deleting Variables

Variables can be deleted by using the **rm()** function. Below we delete the variable var.3. On printing the value of the variable error is thrown.

```
rm(var.3)

print(var.3)
```

When we execute the above code, it produces the following result –

```
[1] "var.3"
Error in print(var.3) : object 'var.3' not found
```

All the variables can be deleted by using the **rm()** and **ls()** function together.

```
rm(list = ls())

print(ls())
```

When we execute the above code, it produces the following result –

```
character(0)
```

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

Types of Operators

We have the following types of operators in R programming –

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

Arithmetic Operators

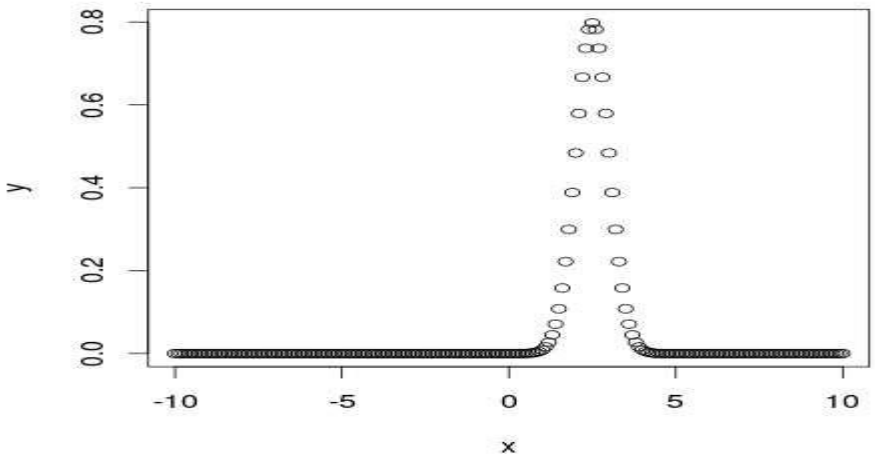
Following table shows the arithmetic operators supported by R language. The operators act on each element of the vector.

```
v <- c( 2,5.5,6)
t <- c(8, 3, 4)
print(v+t)
```

Executing the program....

$Rscript main.r

```
[1] 10.0  8.5 10.0
```



Operator	Description	Example

+	Adds two vectors	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v+t)`
		it produces the following result –
		`[1] 10.0  8.5  10.0`

–	Subtracts second vector from the first	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v-t)`
		it produces the following result –
		`[1] -6.0  2.5  2.0`

*	Multiplies both vectors	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v*t)`
		it produces the following result –
		`[1] 16.0 16.5 24.0`

/	Divide the first vector with the second	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v/t)`
		When we execute the above code, it produces the following result –
		`[1] 0.250000 1.833333 1.500000`

%%	Give the remainder of the first	`v <- c( 2,5.5,6)` `t <- c(8, 3, 4)`

	vector with the second	`print(v%%t)`
		it produces the following result –
		`[1] 2.0 2.5 2.0`

%/%	The result of division of first vector with second (quotient)	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v%/%t)`
		it produces the following result –
		`[1] 0 1 1`

^	The first vector raised to the exponent of second vector	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v^t)`
		it produces the following result –
		`[1]  256.000  166.375 1296.000`

Relational Operators

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example

>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v>t)`
		it produces the following result –
		`[1] FALSE  TRUE FALSE FALSE`

<	Checks if each element of the first vector is less than the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v < t) it produces the following result − [1] TRUE FALSE TRUE FALSE
==	Checks if each element of the first vector is equal to the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v == t) it produces the following result − [1] FALSE FALSE FALSE TRUE
<=	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v<=t) it produces the following result − [1] TRUE FALSE TRUE TRUE
>=	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v>=t) it produces the following result − [1] FALSE TRUE FALSE TRUE

!=	Checks if each element of the first vector is unequal to the corresponding element of the second vector.	v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v!=t) it produces the following result − [1] TRUE TRUE TRUE FALSE

Logical Operators

Following table shows the logical operators supported by R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example
&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.	v <- c(3,1,TRUE,2+3i) t <- c(4,1,FALSE,2+3i) print(v&t) it produces the following result − [1] TRUE TRUE FALSE TRUE
|	It is called Element-wise Logical OR	v <- c(3,0,TRUE,2+2i)

	operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.	`t <- c(4,0,FALSE,2+3i)`
		`print(v|t)`
		it produces the following result –
		`[1]  TRUE FALSE  TRUE  TRUE`

!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.	`v <- c(3,0,TRUE,2+2i)`
		`print(!v)`
		it produces the following result –
		`[1] FALSE  TRUE FALSE FALSE`

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

Operator	Description	Example

&&	Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	`v <- c(3,0,TRUE,2+2i)`
		`t <- c(1,3,TRUE,2+3i)`
		`print(v&&t)`
		it produces the following result –
		`[1] TRUE`

||	Called Logical OR operator. Takes first element of both the vectors and gives the TRUE if one of them is TRUE.	v <- c(0,0,TRUE,2+2i) t <- c(0,3,TRUE,2+3i) print(v||t) it produces the following result – [1] FALSE

Assignment Operators

These operators are used to assign values to vectors.

Operator	Description	Example
<– or = or <<–	Called Left Assignment	v1 <- c(3,1,TRUE,2+3i) v2 <<- c(3,1,TRUE,2+3i) v3 = c(3,1,TRUE,2+3i) print(v1) print(v2) print(v3) it produces the following result – [1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i
–> or –>>	Called Right Assignment	c(3,1,TRUE,2+3i) -> v1 c(3,1,TRUE,2+3i) ->> v2 print(v1) print(v2) it produces the following result – [1] 3+0i 1+0i 1+0i 2+3i

		`[1] 3+0i 1+0i 1+0i 2+3i`

Miscellaneous Operators

These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example

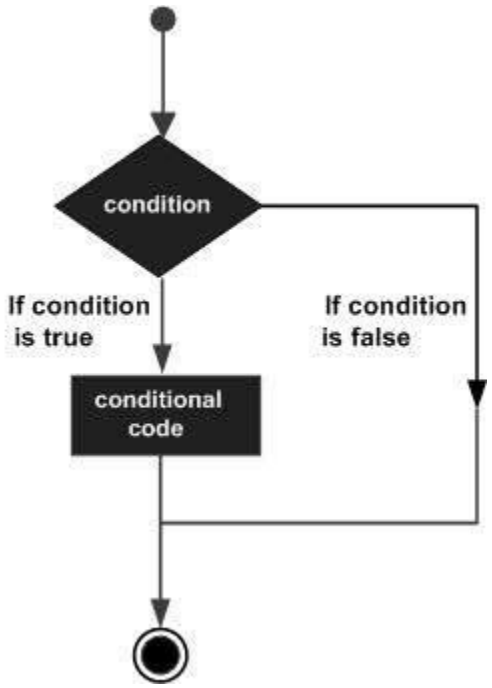
:	Colon operator. It creates the series of numbers in sequence for a vector.	`v <- 2:8`
		`print(v)`
		it produces the following result –
		`[1] 2 3 4 5 6 7 8`

%in%	This operator is used to identify if an element belongs to a vector.	`v1 <- 8`
		`v2 <- 12`
		`t <- 1:10`
		`print(v1 %in% t)`
		`print(v2 %in% t)`
		it produces the following result –
		`[1] TRUE`
		`[1] FALSE`

%*%	This operator is used to multiply a matrix with its transpose.	`M = matrix( c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE)`
		`t = M %*% t(M)`
		`print(t)`
		it produces the following result –
		`[,1] [,2]`
		`[1,] 65 82`
		`[2,] 82 117`

Decision making structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**.

Following is the general form of a typical decision making structure found in most of the programming languages –



R provides the following types of decision making statements. Click the following links to check their detail.

An **if** statement consists of a Boolean expression followed by one or more statements.

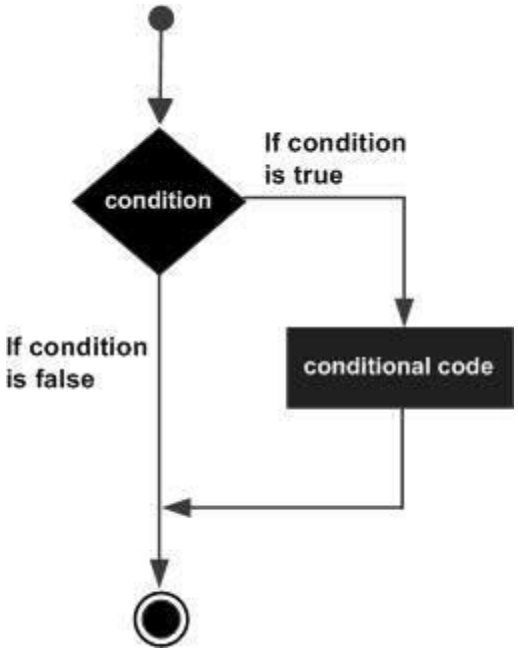
Syntax

The basic syntax for creating an **if** statement in R is –

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
}
```

If the Boolean expression evaluates to be **true**, then the block of code inside the if statement will be executed. If Boolean expression evaluates to be **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Example

```
x <- 30L

if(is.integer(x)) {

   print("X is an Integer")

}
```

When the above code is compiled and executed, it produces the following result –

```
[1] "X is an Integer"
```

```
x <- c("what","is","truth")


if("Truth" %in% x) {

   print("Truth is found")

} else {

   print("Truth is not found")

}
```

When the above code is compiled and executed, it produces the following result –

```
[1] "Truth is not found"
```

Here "Truth" and "truth" are two different strings.

The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using **if**, **else if**, **else** statements there are few points to keep in mind.

- An **if** can have zero or one **else** and it must come after any **else if**'s.
- An **if** can have zero to many **else if's** and they must come before the else.
- Once an **else if** succeeds, none of the remaining **else if**'s or **else**'s will be tested.

Syntax

The basic syntax for creating an **if...else if...else** statement in R is –

```
if(boolean_expression 1) {
   // Executes when the boolean expression 1 is true.
} else if( boolean_expression 2) {
   // Executes when the boolean expression 2 is true.
} else if( boolean_expression 3) {
   // Executes when the boolean expression 3 is true.
} else {
   // executes when none of the above condition is true.
}
```

Example

```
x <- c("what","is","truth")


if("Truth" %in% x) {
   print("Truth is found the first time")
} else if ("truth" %in% x) {
   print("truth is found the second time")
} else {
   print("No truth found")
```

```
}
```

When the above code is compiled and executed, it produces the following result –

```
[1] "truth is found the second time"
```

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

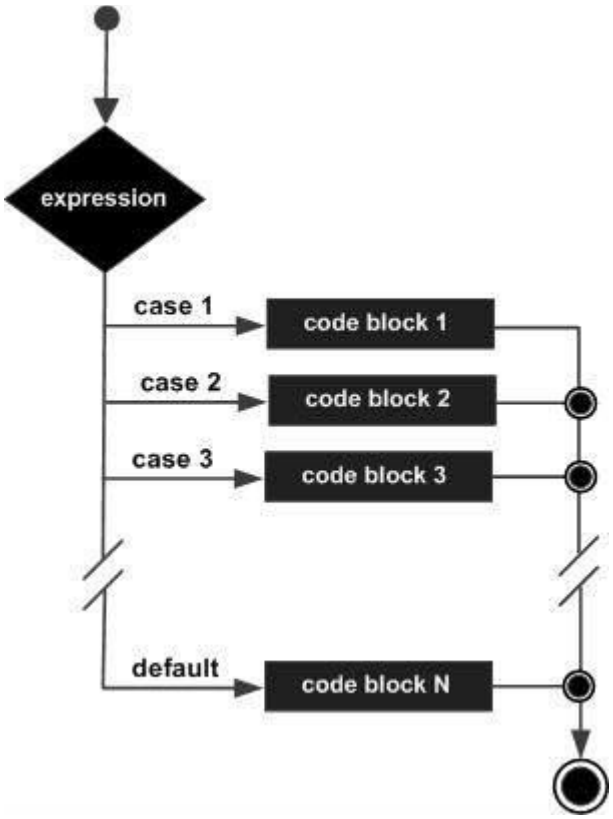
The basic syntax for creating a switch statement in R is –

```
switch(expression, case1, case2, case3....)
```

The following rules apply to a switch statement –

- If the value of expression is not a character string it is coerced to integer.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- If the value of the integer is between 1 and nargs()−1 (The max number of arguments)then the corresponding element of case condition is evaluated and the result returned.
- If expression evaluates to a character string then that string is matched (exactly) to the names of the elements.
- If there is more than one match, the first matching element is returned.
- No Default argument is available.
- In the case of no match, if there is a unnamed element of ... its value is returned. (If there is more than one such argument an error is returned.)

Flow Diagram



Example

```
x <- switch(
   3,
   "first",
   "second",
   "third",
   "fourth"
)
print(x)
```

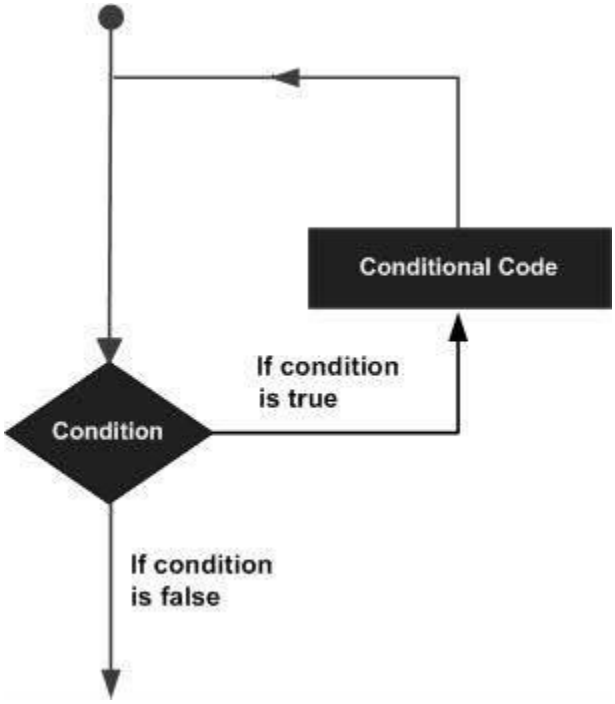
When the above code is compiled and executed, it produces the following result –

```
[1] "third"
```

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and the following is the general form of a loop statement in most of the programming languages −



R programming language provides the following kinds of loop to handle looping requirements. Click the following links to check their detail.

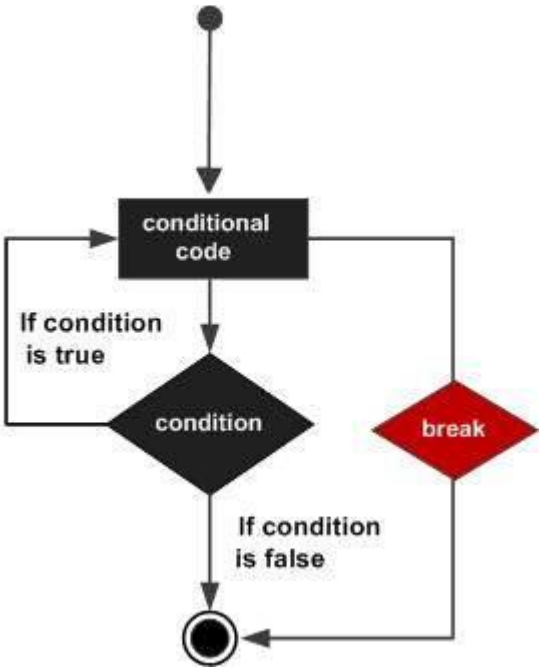
The **Repeat loop** executes the same code again and again until a stop condition is met.

Syntax

The basic syntax for creating a repeat loop in R is −

```
repeat {
   commands
   if(condition) {
      break
   }
}
```

Flow Diagram



Data Type:

Logical

Example TRUE , FALSE

```
v <- TRUE
print(class(v))
```

it produces the following result:

```
[1] "logical"
```

Numeric:

Example 12.3, 5, 999

```
v <- 23.5
print(class(v))
```

it produces the following result:

```
[1] "numeric"
```

Integer:

Example 2L, 34L, 0L

```
v <- 2L
print(class(v))
```

it produces the following result:

```
[1] "integer"
```

Complex:

Example 3 + 2i

```
v <- 2+5i
print(class(v))
```

it produces the following result:

```
[1] "complex"
```

Character

Example 'a' , '"good", "TRUE", '23.4'

```
v <- "TRUE"
print(class(v))
```

it produces the following result:

```
[1] "character"
```

Raw:

"Hello" is stored as 48 65 6c 6c 6f

```
v <- charToRaw("Hello")
print(class(v))
```

it produces the following result:

```
[1] "raw"
```

In R programming, the very basic data types are the R-objects called **vectors** which hold
elements of different classes as shown above. Please note in R the number of classes is
not confined to only the above six types. For example, we can use many atomic vectors
and create an array whose class will become array

Vectors

When you want to create vector with more than one element, you should use **c()** function
which means to combine the elements into a vector.

```
# Create a vector.
apple <- c('red','green',"yellow")
print(apple)
# Get the class of the vector.
print(class(apple))
```

When we execute the above code, it produces the following result:

```
[1] "red" "green" "yellow"
[1] "character"
```

Lists

A list is an R-object which can contain many different types of elements inside it like
vectors, functions and even another list inside it.

```
 # Create a list.
list1 <- list(c(2,5,3),21.3,sin)
# Print the list.
print(list1)
```

When we execute the above code, it produces the following result:

```
[[1]]
[1] 2 5 3
[[2]]
[1] 21.3
[[3]]
function (x) .Primitive("sin")
```

Matrices

A matrix is a two-dimensional rectangular data set. It can be created using a vector input
to the matrix function.

```
# Create a matrix.
M = matrix( c('a','a','b','c','b','a'), nrow=2,ncol=3,byrow = TRUE)
print(M)
```

When we execute the above code, it produces the following result:

```
[,1] [,2] [,3]
```



```
[1,] "a" "a" "b"
[2,] "c" "b" "a"
```

Arrays

While matrices are confined to two dimensions, arrays can be of any number of
dimensions. The array function takes a dim attribute which creates the required number
of dimension. In the below example we create an array with two elements which are 3x3
matrices each.

```
# Create an array.
a <- array(c('green','yellow'),dim=c(3,3,2))
print(a)
```

When we execute the above code, it produces the following result:

```
, , 1
[,1] [,2] [,3]
[1,] "green" "yellow" "green"
[2,] "yellow" "green" "yellow"
```

```
[3,] "green" "yellow" "green"
, , 2
[,1] [,2] [,3]
[1,] "yellow" "green" "yellow"
[2,] "green" "yellow" "green"
[3,] "yellow" "green" "yellow"
```

Factors

Factors are the r-objects which are created using a vector. It stores the vector along with the distinct values of the elements in the vector as labels. The labels are always character irrespective of whether it is numeric or character or Boolean etc. in the input vector. They are useful in statistical modeling.

Factors are created using the **factor()** function.The **nlevels** functions gives the count of levels.

```
# Create a vector.
apple_colors <- c('green','green','yellow','red','red','red','green')
```



```
# Create a factor object.
factor_apple <- factor(apple_colors)
# Print the factor.
print(factor_apple)
print(nlevels(factor_apple))
```

When we execute the above code, it produces the following result:

```
[1] green green yellow red red red yellow green
Levels: green red yellow
# applying the nlevels function we can know the number of distinct
values
[1] 3
```

Data Frames

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.

Data Frames are created using the **data.frame()** function.

```
# Create the data frame.
BMI <- data.frame(
gender = c("Male", "Male","Female"),
height = c(152, 171.5, 165),
weight = c(81,93, 78),
```

```
Age =c(42,38,26)
)
print(BMI)
```

When we execute the above code, it produces the following result:

```
gender height weight Age
1 Male 152.0 81 42
2 Male 171.5 93 38
3 Female 165.0 78 26
```