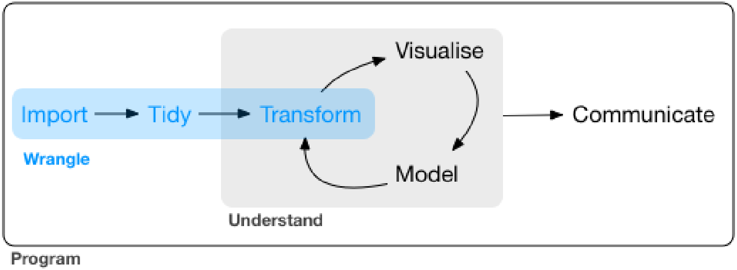
Data Science in Business Analytics

Data Wrangling – 2

HEC Lausanne
Professor Alex [aleksandr.shemendyuk@unil.ch]

Today


Visualise
Import
Tidy
Transform
Communicate
Wrangle
Model
Understand
Program

Outline

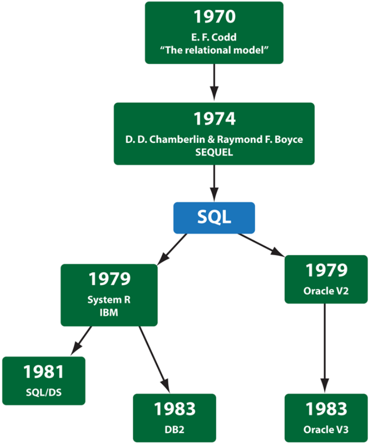
Agenda

Relational Data

- **Until now**: Analysis of a single table of data.
- **Typically**: In practice, data often spans multiple tables that need to be combined.
- **Definition**: This is known as *relational data*:
  - The relationships between tables, not just the individual datasets, are key.
- **Relations**:
  - Defined between pairs of tables.
  - Relationships involving three or more tables are constructed from these pairwise relations.

Relational Database Systems

- Common relational database systems include:
  - Oracle, MySQL, Microsoft SQL Server, PostgreSQL, IBM DB2, Microsoft Access, SQLite, and others.



Datasets Used from `nycflights13`

- In this section, we introduce the datasets from the `nycflights13` package, which will be used for exploring relational data operations.
- These datasets include information on
  - flights,
  - airlines,
  - airports,
  - planes,
  - and weather data.

nycflights13::flights

- Contains 336'776 flights that departed from NYC in 2013:

```
flights
#> # A tibble: 336,776 x 19
#>     year month   day dep_time sched_dep_time dep_delay arr_time
#>    <int> <int> <int>    <int>          <int>     <dbl>    <int>
#>  1  2013     1     1      517            515         2      830
#>  2  2013     1     1      533            529         4      850
#>  3  2013     1     1      542            540         2      923
#>  4  2013     1     1      544            545        -1     1004
#>  5  2013     1     1      554            600        -6      812
#>  6  2013     1     1      554            558        -4      740
#>  7  2013     1     1      555            600        -5      913
#>  8  2013     1     1      557            600        -3      709
#>  9  2013     1     1      557            600        -3      838
#> 10  2013     1     1      558            600        -2      753
#> # i 336,766 more rows
#> # i 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>
```

nycflights13::airlines

Displays airline information corresponding to each carrier in the `flights` dataset:

```
airlines
#> # A tibble: 16 x 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.
#>  7 F9      Frontier Airlines Inc.
#>  8 FL      AirTran Airways Corporation
#>  9 HA      Hawaiian Airlines Inc.
#> 10 MQ      Envoy Air
#> 11 OO      SkyWest Airlines Inc.
#> 12 UA      United Air Lines Inc.
#> 13 US      US Airways Inc.
#> 14 VX      Virgin America
#> 15 WN      Southwest Airlines Co.
#> 16 YV      Mesa Airlines Inc.
```

nycflights13::airports

Provides details about each airport, including geographic location and other identifying information:

```
airports
#> # A tibble: 1,458 x 8
#>    faa   name                           lat    lon   alt    tz dst   tzone
#>    <chr> <chr>                        <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
#>  1 04G   Lansdowne Airport             41.1  -80.6  1044    -5 A     Amer~
#>  2 06A   Moton Field Municipal ~       32.5  -85.7   264    -6 A     Amer~
#>  3 06C   Schaumburg Regional           42.0  -88.1   801    -6 A     Amer~
#>  4 06N   Randall Airport               41.4  -74.4   523    -5 A     Amer~
#>  5 09J   Jekyll Island Airport         31.1  -81.4    11    -5 A     Amer~
#>  6 0A9   Elizabethton Municipal~       36.4  -82.2  1593    -5 A     Amer~
#>  7 0G6   Williams County Airport       41.5  -84.5   730    -5 A     Amer~
#>  8 0G7   Finger Lakes Regional ~       42.9  -76.8   492    -5 A     Amer~
#>  9 0P2   Shoestring Aviation Ai~       39.8  -76.6  1000    -5 U     Amer~
#> 10 0S9   Jefferson County Intl         48.1 -123.    108    -8 A     Amer~
#> # i 1,448 more rows
```

nycflights13::planes

Contains information on the planes used in the flights, including the manufacturer and year built:

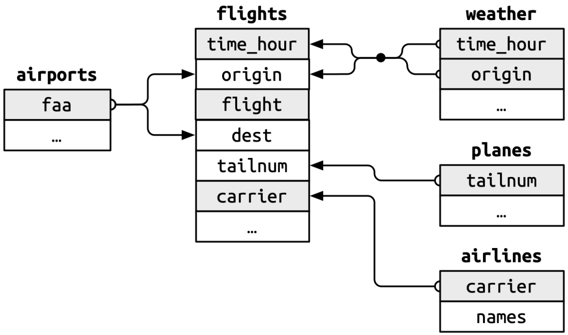
```
planes
#> # A tibble: 3,322 x 9
#>    tailnum  year type   manufacturer model engines seats speed engine
#>    <chr>   <int> <chr>  <chr>        <chr>   <int> <int> <int> <chr>
#>  1 N10156   2004 Fixed~ EMBRAER      EMB-~       2    55    NA Turbo~
#>  2 N102UW   1998 Fixed~ AIRBUS INDU~ A320~       2   182    NA Turbo~
#>  3 N103US   1999 Fixed~ AIRBUS INDU~ A320~       2   182    NA Turbo~
#>  4 N104UW   1999 Fixed~ AIRBUS INDU~ A320~       2   182    NA Turbo~
#>  5 N10575   2002 Fixed~ EMBRAER      EMB-~       2    55    NA Turbo~
#>  6 N105UW   1999 Fixed~ AIRBUS INDU~ A320~       2   182    NA Turbo~
#>  7 N107US   1999 Fixed~ AIRBUS INDU~ A320~       2   182    NA Turbo~
#>  8 N108UW   1999 Fixed~ AIRBUS INDU~ A320~       2   182    NA Turbo~
#>  9 N109UW   1999 Fixed~ AIRBUS INDU~ A320~       2   182    NA Turbo~
#> 10 N110UW   1999 Fixed~ AIRBUS INDU~ A320~       2   182    NA Turbo~
#> # i 3,312 more rows
```

nycflights13::weather

Records weather conditions in NYC airports at the time of each flight's departure:

```
weather
#> # A tibble: 26,115 x 15
#>    origin  year month   day  hour  temp  dewp humid wind_dir
#>    <chr>  <int> <int> <int> <int> <dbl> <dbl> <dbl>    <dbl>
#>  1 EWR     2013     1     1     1  39.0  26.1  59.4      270
#>  2 EWR     2013     1     1     2  39.0  27.0  61.6      250
#>  3 EWR     2013     1     1     3  39.0  28.0  64.4      240
#>  4 EWR     2013     1     1     4  39.9  28.0  62.2      250
#>  5 EWR     2013     1     1     5  39.0  28.0  64.4      260
#>  6 EWR     2013     1     1     6  37.9  28.0  67.2      240
#>  7 EWR     2013     1     1     7  39.0  28.0  64.4      240
#>  8 EWR     2013     1     1     8  39.9  28.0  62.2      250
#>  9 EWR     2013     1     1     9  39.9  28.0  62.2      260
#> 10 EWR     2013     1     1    10  41    28.0  59.6      260
#> # i 26,105 more rows
#> # i 6 more variables: wind_speed <dbl>, wind_gust <dbl>,
#> #   precip <dbl>, pressure <dbl>, visib <dbl>, time_hour <dttm>
```

nycflights13: Relationships Summary



Keys: Primary and Foreign

Keys connect two tables by linking variables.

- **Primary Key**: Uniquely identifies each observation within its own table.
  - Examples:
    - `airlines$carrier` in `airlines`.
    - `airports$faa` in `airports`.
    - `planes$tailnum` in `planes`.
    - Compound keys: `weather$origin` and `weather$time_hour` in `weather`.

Foreign Keys and Table Relationships

- **Foreign Key**: Links to a primary key in another table.
  - Examples:
    - `flights$tailnum` links to `planes$tailnum`.
    - `flights$carrier` links to `airlines$carrier`.
    - Compound keys like `flights$origin` and `flights$time_hour` link to `weather$origin` and `weather$time_hour`.
- **Naming Convention**: Primary and foreign keys often share names, simplifying table joins.

Verifying Primary Keys

To confirm that a key is primary, we check:

- **Uniqueness**: Each key should uniquely identify an observation.

```
planes |>
    count(tailnum) |>
    filter(n > 1)
#> # A tibble: 0 x 2
#> # i 2 variables: tailnum <chr>, n <int>
```

- **No Missing Values**: A primary key must not contain `NA` values.

```
planes |>
    filter(is.na(tailnum))
#> # A tibble: 0 x 9
#> # i 9 variables: tailnum <chr>, year <int>, type <chr>,
#> #   manufacturer <chr>, model <chr>, engines <int>, seats <int>,
#> #   speed <int>, engine <chr>
```

Surrogate Keys

- **Complex Keys**: Some tables lack a simple primary key, requiring a combination of variables to uniquely identify rows.
  - For `flights`, the combination `time_hour`, `carrier`, and `flight` uniquely identifies each observation.

```
flights |>
    count(time_hour, carrier, flight) |>
    filter(n > 1)
#> # A tibble: 0 x 4
#> # i 4 variables: time_hour <dttm>, carrier <chr>, flight <int>,
#> #   n <int>
```

Creating a Surrogate Key

- **Surrogate Key**: A simpler, unique identifier can be generated to make referencing observations easier.
  - Adding a numeric ID as a surrogate key for each row in `flights`:

```
flights2 <- flights |>
  mutate(id = row_number(), .before = 1)
```

```
#> # A tibble: 336,776 x 20
#>      id  year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1     1  2013     1     1      517            515         2      830
#> 2     2  2013     1     1      533            529         4      850
#> 3     3  2013     1     1      542            540         2      923
#> 4     4  2013     1     1      544            545        -1     1004
#> 5     5  2013     1     1      554            600        -6      812
#> # i 336,771 more rows
#> # i 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>
```

Combining Tables

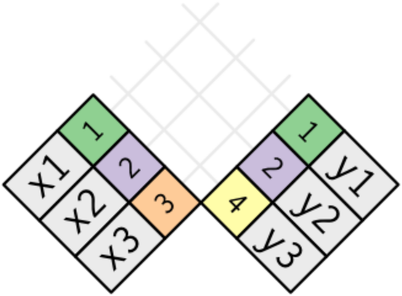
- Two main types of verbs for working with relational data:
  - **Mutating Joins**: Add new variables to a data frame based on matching observations in another.
  - **Filtering Joins**: Filter observations in a data frame depending on whether they match observations in another table.

Technical Slide: Narrow `flights` Dataset

```
flights2 <- flights |>
    select(year:day, hour, origin, dest, tailnum, carrier)

flights2
#> # A tibble: 336,776 x 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
#>  7  2013     1     1     6 EWR    FLL   N516JB  B6
#>  8  2013     1     1     6 LGA    IAD   N829AS  EV
#>  9  2013     1     1     6 JFK    MCO   N593JB  B6
#> 10  2013     1     1     6 LGA    ORD   N3ALAA  AA
#> # i 336,766 more rows
```

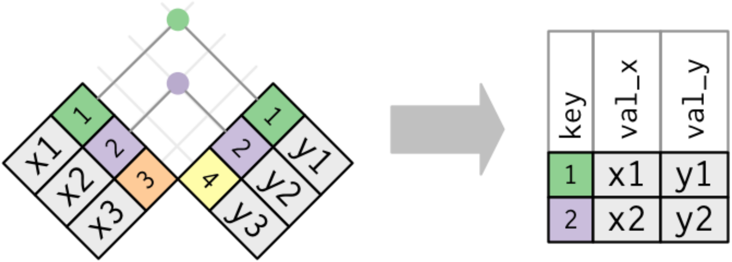
Understanding Mutating Joins

```
x <- tribble(~key, ~val_x,
             1, "x1",
             2, "x2",
             3, "x3")
y <- tribble(~key, ~val_y,
             1, "y1",
             2, "y2",
             4, "y3")
```



Inner Join

An **inner join** retains only matching rows between tables `x` and `y`.




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

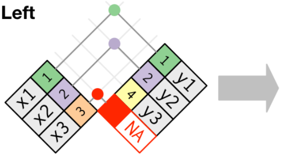
Outer Joins

- **Outer Joins** retain observations appearing in at least one of the tables:
  - **Left join**: Keeps all observations from `x`.
  - **Right join**: Keeps all observations from `y`.
  - **Full join**: Keeps all observations from both `x` and `y`.

Outer joins introduce "virtual" observations with keys that match all unmatched rows, with missing values filled as `NA`.

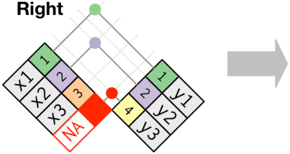
Outer Joins Illustrated

Left



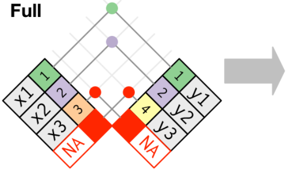
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

Right



key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3

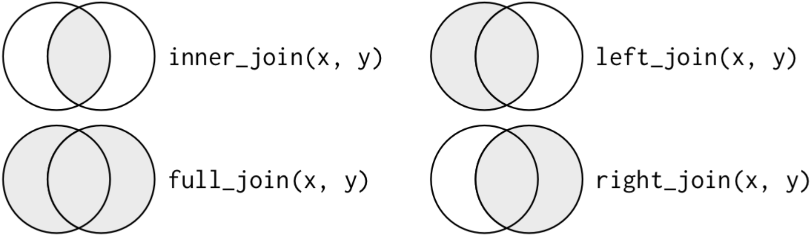
Full



key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

A Venn Diagram for Joins

This Venn diagram visually summarizes inner, left, right, and full joins:



Duplicate Keys

- When tables contain duplicate keys, two scenarios arise:
  - **One table has duplicate keys**:
    - Common in one-to-many relationships where additional information is added.
  - **Both tables have duplicate keys**:
    - Typically an error as keys no longer uniquely identify observations.
    - Joins produce all possible combinations (Cartesian product) when both tables have duplicate keys.

One Table with Duplicate Keys

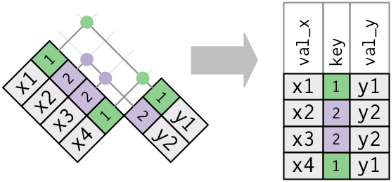
- Only x contains duplicated keys:

```
x <- tribble(~key, ~val_x,
             1, "x1",
             2, "x2",
             2, "x3",
             1, "x4")
```

```
y <- tribble(~key, ~val_y,
             1, "y1",
             2, "y2")
```

- Resulting join adds val_y to matching rows:

```
left_join(x, y, join_by(key))
#> # A tibble: 4 x 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
```



Both Tables with Duplicate Keys

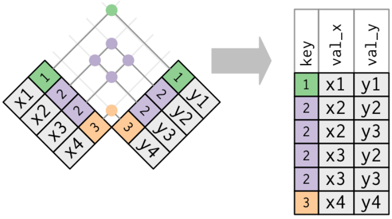
- Both x and y contain duplicated keys:

```r
x <- tribble(~key, ~val_x,
             1, "x1",
             2, "x2",
             2, "x3",
             3, "x4")
```

```r
y <- tribble(~key, ~val_y,
             1, "y1",
             2, "y2",
             2, "y3",
             3, "y4")
```

- Join results in all possible combinations:

```r
left_join(x, y, join_by(key))
#> # A tibble: 6 x 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
```



Specifying the Keys

- By default, `left_join()` uses all variables common to both tables as the join keys, performing what is called a **natural join**.
  - This is convenient but may not always produce the intended result.

```
flights2 |>
  left_join(weather) |> print(n = 5)
#> # A tibble: 336,776 x 18
#>    year month   day  hour origin dest  tailnum carrier  temp  dewp
#>   <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <dbl> <dbl>
#> 1  2013     1     1     5 EWR    IAH   N14228  UA       39.0  28.0
#> 2  2013     1     1     5 LGA    IAH   N24211  UA       39.9  25.0
#> 3  2013     1     1     5 JFK    MIA   N619AA  AA       39.0  27.0
#> 4  2013     1     1     5 JFK    BQN   N804JB  B6       39.0  27.0
#> 5  2013     1     1     6 LGA    ATL   N668DN  DL       39.9  25.0
#> # i 336,771 more rows
#> # i 8 more variables: humid <dbl>, wind_dir <dbl>, wind_speed <dbl>,
#> #   wind_gust <dbl>, precip <dbl>, pressure <dbl>, visib <dbl>,
#> #   time_hour <dttm>
```

Specifying Specific Join Keys

- Sometimes only a subset of the common variables should be used for joining.
  - In this example, we join `flights2` with `planes` using `tailnum` only, avoiding unintended matches on other common variables like `year`.

```
flights2 |>
    left_join(planes, join_by(tailnum)) |> print(n = 5)
#> # A tibble: 336,776 x 16
#>   year.x month   day  hour origin dest  tailnum carrier year.y type
#>    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>    <int> <chr>
#> 1   2013     1     1     5 EWR    IAH   N14228  UA        1999 Fixed~
#> 2   2013     1     1     5 LGA    IAH   N24211  UA        1998 Fixed~
#> 3   2013     1     1     5 JFK    MIA   N619AA  AA        1990 Fixed~
#> 4   2013     1     1     5 JFK    BQN   N804JB  B6        2012 Fixed~
#> 5   2013     1     1     6 LGA    ATL   N668DN  DL        1991 Fixed~
#> # i 336,771 more rows
#> # i 6 more variables: manufacturer <chr>, model <chr>,
#> #   engines <int>, seats <int>, speed <int>, engine <chr>
```

- Notice that the `year` columns are disambiguated with suffixes (`year.x` and `year.y`) to indicate the origin table.

Custom Join Conditions

- We can specify joins with different variable names using `join_by(a == b)`.
  - Here, `left_join` matches `dest` in `flights2` with `faa` in `airports`.

```
flights2 |>
  left_join(airports, join_by(dest == faa)) |> print(n = 5)
#> # A tibble: 336,776 x 15
#>    year month   day  hour origin dest  tailnum carrier name         lat
#>   <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <chr>      <dbl>
#> 1  2013     1     1     5 EWR    IAH   N14228  UA      George ~  30.0
#> 2  2013     1     1     5 LGA    IAH   N24211  UA      George ~  30.0
#> 3  2013     1     1     5 JFK    MIA   N619AA  AA      Miami I~  25.8
#> 4  2013     1     1     5 JFK    BQN   N804JB  B6      <NA>      NA
#> 5  2013     1     1     6 LGA    ATL   N668DN  DL      Hartsfi~  33.6
#> # i 336,771 more rows
#> # i 5 more variables: lon <dbl>, alt <dbl>, tz <dbl>, dst <chr>,
#> #   tzone <chr>
```

- This approach clarifies which keys are matched and supports more complex join requirements.

Filtering Joins

Filtering joins affect the rows, not the columns:

- `semi_join(x, y)`: Keeps all rows in `x` that have a match in `y`.
  - Useful for filtering to matching observations in both tables.
- `anti_join(x, y)`: Drops all rows in `x` that have a match in `y`.
  - Useful for diagnosing mismatches, identifying records in `x` without a corresponding match in `y`.

Flights to Top Destinations

To identify flights to top destinations, filter the `flights` data for destinations with high frequency:

```
top_dest <- flights |>
    count(dest, sort = TRUE) |>
    head(10)

flights |>
    filter(dest %in% top_dest$dest) |>
    print(n = 5)
#> # A tibble: 141,145 x 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
#> # i 141,140 more rows
#> # i 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>
```

Using Semi-Join

- `semi_join()` retains only rows in `flights` that match with rows in `top_dest`:

```
semi_join(flights, top_dest, join_by(dest))
#> # A tibble: 141,145 x 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
#>  7  2013     1     1      558            600        -2      924
#>  8  2013     1     1      558            600        -2      923
#>  9  2013     1     1      559            559         0      702
#> 10  2013     1     1      600            600         0      851
#> # i 141,135 more rows
#> # i 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>
```

Using Anti-Join

- `anti_join()` helps identify rows in `flights2` without matches in other tables, making it useful for detecting mismatches.

- Finding destinations in `flights2` that are not listed in `airports`:

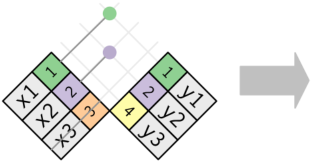
```
flights2 |>
  anti_join(
    airports,
    join_by(dest == faa)
  ) |>
  distinct(dest)
#> # A tibble: 4 x 1
#>   dest
#>   <chr>
#> 1 BQN
#> 2 SJU
#> 3 STT
#> 4 PSE
```

- Finding tail numbers in `flights2` that are not present in `planes`:

```
flights2 |>
  anti_join(planes, join_by(tailnum)) |>
  distinct(tailnum) |> print(n = 5)
#> # A tibble: 722 x 1
#>   tailnum
#>   <chr>
#> 1 N3ALAA
#> 2 N3DUAA
#> 3 N542MQ
#> 4 N730MQ
#> 5 N9EAMQ
#> # i 717 more rows
```

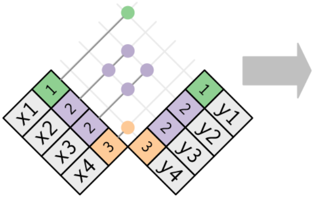
Visually Understanding the Semi-Join

- **One-to-Many**: A semi-join between tables with a one-to-many relationship.



key	val_x
1	x1
2	x2

- **Many-to-Many**: A semi-join between tables with a many-to-many relationship.

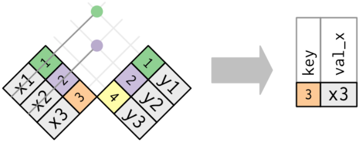


key	val_x
1	x1
2	x2
2	x3
3	x4

`flights` Without a Match in `planes`

Using `anti_join()` to identify tail numbers in `flights` without matches in `planes`.

```
flights |>
  anti_join(planes, join_by(tailnum)) |>
  count(tailnum, sort = TRUE)
#> # A tibble: 722 x 2
#>    tailnum     n
#>    <chr>   <int>
#>  1 <NA>     2512
#>  2 N725MQ    575
#>  3 N722MQ    513
#>  4 N723MQ    507
#>  5 N713MQ    483
#>  6 N735MQ    396
#>  7 N0EGMQ    371
#>  8 N534MQ    364
#>  9 N542MQ    363
#> 10 N531MQ    349
#> # i 712 more rows
```



Agenda

Warm-Up: Rethinking Time

Let's start with a few quick questions about time:

- **Does every year have exactly 365 days?**
  - *Hint*: What happens every four years?
- **Does every day always have 24 hours?**
  - *Hint*: Consider daylight saving time adjustments.
- **Does every minute have 60 seconds?**
  - *Hint*: Some "special" minutes may surprise you!

Exploring dates and times might seem simple at first, but they can involve surprising complexity. Let's dive in and see why!

Referring to an Instant in Time

- **Two Types of Date/Time Data**:
  - **Date**: Represents a specific day (printed as `<date>` in tibbles).
  - **Date-Time**: Combines a date with a time to specify a precise instant (printed as `<dttm>` in tibbles).
    - Equivalent to `POSIXct` in base R.
    - Use date-times only when necessary as they are more complex due to time zones.
- **Tip**: Always use the simplest possible data type for your needs.

Creating Dates and Date-Times

- The **`lubridate`** package simplifies working with dates and times in R.
  - As of `tidyverse 2.0.0`, `lubridate` is part of the core tidyverse.

```
library(lubridate)
today()    # Current date
#> [1] "2024-11-03"
now()      # Current date-time
#> [1] "2024-11-03 23:38:49 CET"
```

Additional Ways to Create Date/Times

- Common methods for creating date/time objects:
  - **From a string**.
  - **From individual date/time components**.
  - **From an existing date/time object**.

```
as_datetime(today())  # Convert date to date-time
#> [1] "2024-11-03 UTC"
as_date(now())        # Convert date-time to date
#> [1] "2024-11-03"
```

Importing Dates and Date-Times

- **Automatic Parsing**: If a CSV file contains dates or date-times in ISO8601 format, `readr` will automatically detect them.

```
csv <- "
  date,datetime
  2022-01-02,2022-01-02 05:12
"
read_csv(csv)
#> # A tibble: 1 x 2
#>   date       datetime
#>   <date>     <dttm>
#> 1 2022-01-02 2022-01-02 05:12:00
```

ISO8601 Standard

- **ISO8601**: International format for dates and times, with elements ordered from largest to smallest.
  - Date: `YYYY-MM-DD` (e.g., 2022-05-03)
  - Date-Time:
    - `YYYY-MM-DD HH:MM:SS`
    - `YYYY-MM-DDTHH:MM:SS`
  - Example: 4:26pm on May 3, 2022 as
    - `2022-05-03 16:26`
    - `2022-05-03T16:26`

Custom Date Formats

- For non-ISO8601 formats, specify `col_types` with `col_date()` or `col_datetime()` and a format string.

Code	Meaning	Example
`%Y`	4-digit year	`2021`
`%y`	2-digit year	`21`
`%m`	Month number	`02`
`%b`	Abbreviated month name	`Feb`
`%B`	Full month name	`February`
`%d`	Day (one or two digits)	`2`
`%H`	Hour (24-hour clock)	`13`
`%I`	Hour (12-hour clock)	`1`
`%p`	AM/PM	`pm`
`%M`	Minutes	`35`
`%S`	Seconds	`45`
`%Z`	Time zone name	`America/Chicago`
`%z`	Offset from UTC	`+0800`

Specifying Ambiguous Date Formats

- For ambiguous formats, use specific format strings:

```
csv <- "
  date
  01/02/15
"
#> Interprets as "2015-01-02"
read_csv(csv, col_types = cols(date = col_date("%m/%d/%y")))
#> # A tibble: 1 x 1
#>   date
#>   <date>
#> 1 2015-01-02

#> Interprets as "2015-02-01"
read_csv(csv, col_types = cols(date = col_date("%d/%m/%y")))
#> # A tibble: 1 x 1
#>   date
#>   <date>
#> 1 2015-02-01
```

- **Locale-Specific Parsing**: Use `locale()` for non-English dates, especially with `%b` or `%B`.

Creating Dates and Date-Times from Strings

- **`lubridate` Helpers**: Use `ymd()`, `mdy()`, `dmy()`, etc., to parse dates automatically based on the order of year, month, and day.

```
ymd("2017-01-31")             # Year-Month-Day
#> [1] "2017-01-31"
mdy("January 31st, 2017")      # Month-Day-Year
#> [1] "2017-01-31"
dmy("31-Jan-2017")             # Day-Month-Year
#> [1] "2017-01-31"
```

- **Creating Date-Times**: Add `_h`, `_m`, `_s` for hour, minute, second as needed.

```
ymd_hms("2017-01-31 20:11:59")  # Year-Month-Day Hour:Minute:Second
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")      # Month-Day-Year Hour:Minute
#> [1] "2017-01-31 08:01:00 UTC"
```

- **Forcing Time Zones**: Specify `tz` to create date-times in a particular timezone, like UTC.

```
ymd(20170131)                    # Interprets as date "2017-01-31"
#> [1] "2017-01-31"
ymd(20170131, tz = "UTC")        # Interprets as date-time in UTC
#> [1] "2017-01-31 UTC"
```

Creating Dates and Date-Times from Components

- Sometimes, date and time components are in separate columns. Use `make_datetime()` to combine them into a single date-time.

```
flights |>
  select(year:day, hour, minute, dep_time) |>
  mutate(departure = make_datetime(year, month, day, hour, minute))
#> # A tibble: 336,776 x 7
#>     year month   day  hour minute dep_time departure
#>    <int> <int> <int> <dbl>  <dbl>    <int> <dttm>
#>  1  2013     1     1     5     15      517 2013-01-01 05:15:00
#>  2  2013     1     1     5     29      533 2013-01-01 05:29:00
#>  3  2013     1     1     5     40      542 2013-01-01 05:40:00
#>  4  2013     1     1     5     45      544 2013-01-01 05:45:00
#>  5  2013     1     1     6      0      554 2013-01-01 06:00:00
#>  6  2013     1     1     5     58      554 2013-01-01 05:58:00
#>  7  2013     1     1     6      0      555 2013-01-01 06:00:00
#>  8  2013     1     1     6      0      557 2013-01-01 06:00:00
#>  9  2013     1     1     6      0      557 2013-01-01 06:00:00
#> 10  2013     1     1     6      0      558 2013-01-01 06:00:00
#> # i 336,766 more rows
```

Handling Special Formats (e.g., dep_time)

- For columns like dep_time (e.g., 531 for 5:31 am), use modulus arithmetic to split hour and minute components.

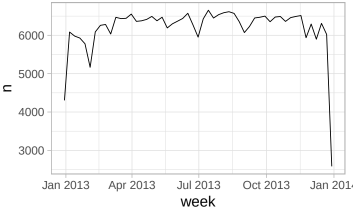
```
flights_dt <- flights |>
  mutate(dep_time = make_datetime(
    year, month, day, dep_time %/% 100, dep_time %% 100))
```

- This approach allows creation of consistent date-time columns from component columns, ready for analysis.

Rounding Dates and Times

- Use rounding functions to simplify date-time values by rounding to a specified unit:
  - **floor_date()**: Rounds down to the nearest unit.
  - **round_date()**: Rounds to the nearest unit.
  - **ceiling_date()**: Rounds up to the nearest unit.

```
flights_dt_plot <- flights_dt |>
  filter(!is.na(dep_time)) |>
  count(
    week = floor_date(dep_time, "week")
  ) |>
  ggplot(aes(week, n)) +
  geom_line()
```



- In this example, `floor_date()` rounds `dep_time` down to the start of each week, making it easy to plot weekly counts of departures.

Getting and Setting Date-Time Components

Extracting Components

- Use accessor functions to get individual parts of a date-time:
  - `year()`, `month()`, `mday()` (day of month), `yday()` (day of year), `wday()` (day of week), `hour()`, `minute()`, `second()`

```
datetime <- ymd_hms("2016-07-08 12:34:56")
c(year(datetime), month(datetime, label = TRUE),
  mday(datetime), yday(datetime))
#> [1] 2016    7    8  190

wday(datetime, label = TRUE, abbr = FALSE)
#> [1] Friday
#> 7 Levels: Sunday < Monday < Tuesday < Wednesday < ... < Saturday
```

Modifying Components

- Use these same functions to adjust components:

```r
year(datetime) <- 2020
month(datetime) <- 1
hour(datetime) <- hour(datetime) + 1
datetime
#> [1] "2020-01-08 13:34:56 UTC"
```

- **Note**: Changing a component will automatically roll over if values exceed their normal limits.

Alternative: Using `update()`

- The `update()` function allows you to modify multiple components at once:

```
update(datetime, year = 2030, month = 2, mday = 2, hour = 2)
#> [1] "2030-02-02 02:34:56 UTC"
```

This approach is useful when multiple updates are needed, ensuring clean and efficient code.

Time Spans

- **Goal**: Perform arithmetic operations (addition, subtraction, division) with dates and times.
- Three main classes for handling time spans in `lubridate`:
  - **Durations**: Represent exact time spans in seconds.
  - **Periods**: Represent human units like days, weeks, or months.
  - **Intervals**: Define a specific time span between a start and end date.

Tip

Choose the simplest class that meets your needs:

- **Physical time** → Duration.
- **Human time units** → Period.
- **Exact time span between points** → Interval.

Durations

- A **duration** always records a time span in seconds, using fixed conversions for larger units:
  - **Conversions**: 60s/minute, 60min/hour, 24h/day, 7d/week, 365.25d/year

```
dseconds(15)
#> [1] "15s"
dminutes(10)
#> [1] "600s (~10 minutes)"
dhours(c(12, 24))
#> [1] "43200s (~12 hours)" "86400s (~1 days)"
ddays(0:5)
#> [1] "0s"                "86400s (~1 days)"  "172800s (~2 days)"
#> [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
dweeks(3)
#> [1] "1814400s (~3 weeks)"
dyears(1)
#> [1] "31557600s (~1 years)"
```

Duration Arithmetic

- **Basic Operations**: Add and multiply durations to combine them flexibly.

```
2 * dyears(1)
#> [1] "63115200s (~2 years)"
dyears(1) + dweeks(12) + dhours(15)
#> [1] "38869200s (~1.23 years)"
```

- **Date Arithmetic**: Add or subtract durations from dates/datetimes.

```
tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)
```

```
#> [1] "2024-11-04"
#> [1] "2023-11-03 18:00:00 UTC"
```

- **Daylight Saving Time Example**: DST can affect exact durations.

```
one_pm <- ymd_hms("2016-03-12 13:00:00", tz = "America/New_York")
one_pm
#> [1] "2016-03-12 13:00:00 EST"
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
```

- Here, adding one day shifts the time due to DST, highlighting the difference between calendar days and fixed-second durations.

Periods

- **Periods** represent time spans in “human” units, like days, months, and years (unlike durations, they don’t have a fixed length in seconds).

```
seconds(15)
#> [1] "15S"
minutes(10)
#> [1] "10M 0S"
hours(c(12, 24))
#> [1] "12H 0M 0S" "24H 0M 0S"
days(7)
#> [1] "7d 0H 0M 0S"
months(1:3)
#> [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S"
weeks(3)
#> [1] "21d 0H 0M 0S"
years(1)
#> [1] "1y 0m 0d 0H 0M 0S"
```

Period Arithmetic

- **Add and multiply** periods to handle flexible time spans.

```
10 * (months(6) + days(1))
#> [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
#> [1] "50d 25H 2M 0S"
```

- **Date-Time Addition**: Periods adapt to calendar-based changes like leap years and daylight saving.

```
# Leap year behavior
ymd("2016-01-01") + dyears(1)  # Fixed duration in seconds
#> [1] "2016-12-31 06:00:00 UTC"
ymd("2016-01-01") + years(1)   # Calendar-aware
#> [1] "2017-01-01"

# Daylight Saving Time adjustment
one_pm + ddays(1)    # Exact duration
#> [1] "2016-03-13 14:00:00 EDT"
one_pm + days(1)     # Period handling
#> [1] "2016-03-13 13:00:00 EDT"
```

Understanding Intervals

- **Intervals** represent a time span with a specified starting and ending point.
  - Helpful for precise calculations that depend on specific dates.
- **Example of Estimation**:
  - Using `years(1) / days(1)` gives an estimated answer based on an average year length of 365.25 days.

```r
years(1) / days(1)
#> [1] 365.25
```

- To achieve a more accurate result, we need an interval.

Creating and Using Intervals

- **Accurate Calculation with Intervals**:
  - Define an interval from today to the same date next year to find the exact number of days:

```
next_year <- today() + years(1)
(today() %--% next_year) / ddays(1)
#> [1] 365
```

- **Leap Year Example**:
  - Calculate days in 2023 vs. 2024 (a leap year):

```
y2023 <- ymd("2023-01-01") %--% ymd("2024-01-01")
y2023
#> [1] 2023-01-01 UTC--2024-01-01 UTC
y2024 <- ymd("2024-01-01") %--% ymd("2025-01-01")

y2023 / days(1)  # Returns 365 for 2023
#> [1] 365
y2024 / days(1)  # Returns 366 for leap year 2024
#> [1] 366
```

Intervals allow precise calculations when calendar variations, such as leap years, affect durations.

Summary

- **Choosing the Right Time Structure**:
  - **Duration**: Use when working with exact time measurements in seconds (e.g., physical elapsed time).
  - **Period**: Use when working with human-centric time spans, such as adding weeks or months.
  - **Interval**: Use when calculating the precise time span between two specific dates, taking into account calendar variations (e.g., leap years).

	date				date time				duration				period				interval				number			
date	-								-	+			-	+							-	+		
date time					-				-	+			-	+							-	+		
duration	-	+			-	+			-	+		/									-	+	×	/
period	-	+			-	+							-	+							-	+	×	/
interval												/				/								
number	-	+			-	+			-	+	×		-	+	×		-	+	×		-	+	×	/

Time Zones

- **Understanding Time Zones**: R uses the IANA time zone database, which identifies time zones by `{continent}/{city}`, e.g., `America/New_York`.
  - Check your system's time zone and see the full list:

```
Sys.timezone()          # Current system time zone
#> [1] "Europe/Berlin"
length(OlsonNames())    # Total number of time zones
#> [1] 596
head(OlsonNames())      # Sample of time zones
#> [1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa"
#> [4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmera"
```

Representing the Same Instant in Different Time Zones

- **Same instant, different time zones**:

```
(x1 <- ymd_hms("2015-06-01 12:00:00", tz = "America/New_York"))
#> [1] "2015-06-01 12:00:00 EDT"
(x2 <- ymd_hms("2015-06-01 18:00:00", tz = "Europe/Copenhagen"))
#> [1] "2015-06-01 18:00:00 CEST"
(x3 <- ymd_hms("2015-06-02 04:00:00", tz = "Pacific/Auckland"))
#> [1] "2015-06-02 04:00:00 NZST"

# Verifying identical instants
x1 - x2  # 0 secs
#> Time difference of 0 secs
x1 - x3  # 0 secs
#> Time difference of 0 secs
```

- **Combining Date-Times**: Using `c()` can unify times to the first element's time zone:

```
x4 <- c(x1, x2, x3)
x4
#> [1] "2015-06-01 12:00:00 EDT" "2015-06-01 12:00:00 EDT"
#> [3] "2015-06-01 12:00:00 EDT"
```

Changing the Time Zone Display

- **Keep the Instant in Time**: Use `with_tz()` to adjust the time zone display without changing the instant.

```
x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
x4a
#> [1] "2015-06-02 02:30:00 +1030" "2015-06-02 02:30:00 +1030"
#> [3] "2015-06-02 02:30:00 +1030"
x4a - x4   # 0 seconds difference
#> Time differences in secs
#> [1] 0 0 0
```

Adjusting the Instant in Time

- **Change the Instant in Time**: Use `force_tz()` when the original time zone is incorrect, altering the actual time.

```
x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
x4b
#> [1] "2015-06-01 12:00:00 +1030" "2015-06-01 12:00:00 +1030"
#> [3] "2015-06-01 12:00:00 +1030"
x4b - x4   # Difference in hours (14.5 in this case)
#> Time differences in hours
#> [1] -14.5 -14.5 -14.5
```

- **Note**: Time zone offsets can vary by non-integer hours, as seen with +1030 for Lord Howe.

Agenda

Introduction to Factors

- **Factors** are used for working with **categorical variables**:
  - Categorical variables have a **fixed, known set of possible values** (e.g., days of the week, levels of satisfaction).
  - Factors allow control over the display and ordering of categorical data, even when it's not alphabetical.
- **The `forcats` Package**:
  - Part of the tidyverse, `forcats` provides a range of helpful functions for creating and manipulating factors.

```
library(forcats)
```

Why Use Factors?

- Factors are particularly useful for:
    - Displaying categories in a **custom order** rather than alphabetical.
    - **Reducing memory usage** when working with large categorical datasets by encoding categories.

This section will explore how to create and work with factors effectively.

Creating Factors

- Imagine a variable recording the month:

```
x1 <- c("Dec", "Apr", "Jan", "Mar")
```

- Using strings to store categorical data can lead to:
    - Typos that go unnoticed.
    - Sorting that's not useful for meaningful categories.

```
sort(x1)  # Alphabetical order, not chronological
#> [1] "Apr" "Dec" "Jan" "Mar"
```

Defining Factor Levels

- First, define the valid **levels** in the correct order:

```
month_levels <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
                  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
```

- Then, create a factor:

```
y1 <- factor(x1, levels = month_levels)
y1
#> [1] Dec Apr Jan Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

sort(y1)       # Sorts in chronological order
#> [1] Jan Mar Apr Dec
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

factor(x1)     # Without specified levels, uses alphabetical order
#> [1] Dec Apr Jan Mar
#> Levels: Apr Dec Jan Mar
```

Handling Typos and Custom Order

- Values not in the specified levels are set to `NA`:

```
x2 <- c("Dec", "Apr", "Jam", "Mar")
y2 <- factor(x2, levels = month_levels)
y2  # "Jam" becomes <NA> due to invalid level
#> [1] Dec  Apr  <NA> Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

- **Custom Orders**:
  - Factor created based on appearance order:

```
factor(x1, levels = unique(x1))  # Custom order by appearance
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar
factor(x1) |>
  fct_inorder()                  # Using forcats helper for order
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar
```

Specifying levels helps avoid errors, ensures meaningful order, and makes factors a powerful tool for categorical data.

Using `forcats::gss_cat`

- **`gss_cat`**: A sample dataset from the General Social Survey, focusing on social, economic, and demographic data.

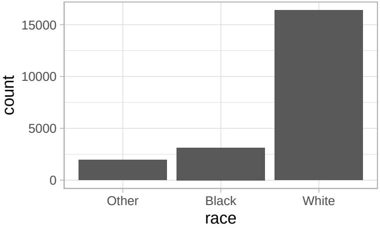
```
gss_cat
#> # A tibble: 21,483 x 9
#>     year marital         age race  rincome partyid relig denom tvhours
#>    <int> <fct>         <int> <fct> <fct>   <fct>   <fct> <fct>   <int>
#>  1  2000 Never marri~     26 White $8000 ~ Ind,ne~ Prot~ Sout~      12
#>  2  2000 Divorced         48 White $8000 ~ Not st~ Prot~ Bapt~      NA
#>  3  2000 Widowed          67 White Not ap~ Indepe~ Prot~ No d~       2
#>  4  2000 Never marri~     39 White Not ap~ Ind,ne~ Orth~ Not ~       4
#>  5  2000 Divorced         25 White Not ap~ Not st~ None  Not ~       1
#>  6  2000 Married          25 White $20000~ Strong~ Prot~ Sout~      NA
#>  7  2000 Never marri~     36 White $25000~ Not st~ Chri~ Not ~       3
#>  8  2000 Divorced         44 White $7000 ~ Ind,ne~ Prot~ Luth~      NA
#>  9  2000 Married          44 White $25000~ Not st~ Prot~ Other       0
#> 10  2000 Married          47 White $25000~ Strong~ Prot~ Sout~       3
#> # i 21,473 more rows
```

- View more details about this dataset with `?gss_cat`.

Viewing Factor Levels in `gss_cat`

- **Plotting Factor Levels**: Create a barplot to visualize distribution.

```
ggplot(gss_cat, aes(race)) +
  geom_bar()
```



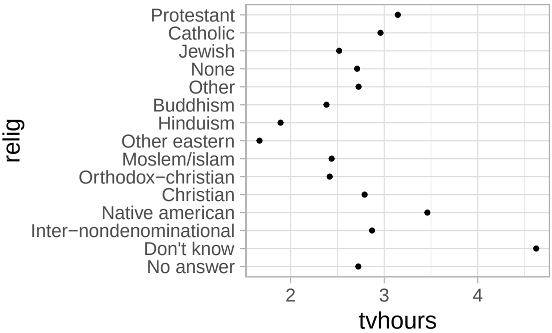
- **Counting Factor Levels**: Use `count()` to summarize factor levels.

```
gss_cat |>
  count(race)
#> # A tibble: 3 x 2
#>   race      n
#>   <fct> <int>
#> 1 Other  1959
#> 2 Black  3129
#> 3 White 16395
```

Troubleshooting Factor Levels in Plots

- What's wrong with this visualization?

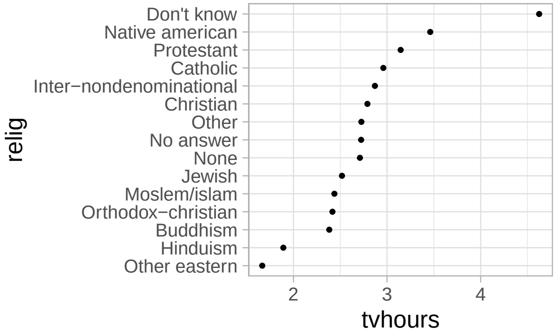
```
relig_summary <- gss_cat |>
  group_by(relig) |>
  summarize(age = mean(age, na.rm = TRUE),
            tvhours = mean(tvhours, na.rm = TRUE),
            n = n())

ggplot(relig_summary, aes(tvhours, relig)) +
  geom_point()
```



Modifying Factor Order

- **Reordering Factors by a Variable**: Use `fct_reorder()` to order factor levels based on another variable's values.

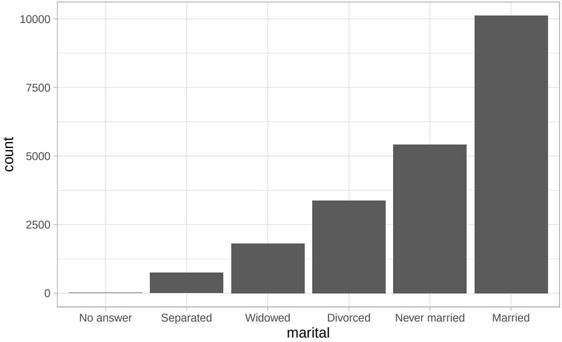
```
relig_summary |>
  mutate(relig = fct_reorder(relig, tvhours)) |>
  ggplot(aes(tvhours, relig)) +
  geom_point()
```



Modifying Factor Order for Frequency

- **Order by Frequency**: Use `fct_infreq()` to order factor levels by their frequency, then reverse with `fct_rev()`.

```
gss_cat |>
  mutate(marital = marital |> fct_infreq() |> fct_rev()) |>
  ggplot(aes(marital)) +
  geom_bar()
```



Modifying Factor Levels

- Beyond reordering, **changing factor levels** allows:
  - **Clarifying labels** for readability or publication.
  - **Collapsing levels** for higher-level summaries.
- Example: Counting the current levels in `partyid`.

```
gss_cat |>
  count(partyid)
#> # A tibble: 10 x 2
#>    partyid                n
#>    <fct>              <int>
#>  1 No answer            154
#>  2 Don't know             1
#>  3 Other party          393
#>  4 Strong republican   2314
#>  5 Not str republican  3032
#>  6 Ind,near rep        1791
#>  7 Independent         4119
#>  8 Ind,near dem        2499
#>  9 Not str democrat    3690
#> 10 Strong democrat     3490
```

Recoding Factor Levels

- **Using** `fct_recode()` to rename levels for clarity:

```
gss_cat |>
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat")) |>
  count(partyid)
#> # A tibble: 10 x 2
#>    partyid                   n
#>    <fct>                 <int>
#>  1 No answer               154
#>  2 Don't know                1
#>  3 Other party             393
#>  4 Republican, strong     2314
#>  5 Republican, weak       3032
#>  6 Independent, near rep  1791
#>  7 Independent            4119
#>  8 Independent, near dem  2499
#>  9 Democrat, weak         3690
#> 10 Democrat, strong       3490
```

Collapsing Factor Levels

- Combine multiple old levels into a single new level using `fct_recode()`:

```
gss_cat |>
  mutate(partyid = fct_recode(partyid,
    "Republican, strong" = "Strong republican",
    "Republican, weak"   = "Not str republican",
    "Other"              = "No answer",
    "Other"              = "Don't know",
    "Other"              = "Other party")) |>
  count(partyid)
#> # A tibble: 8 x 2
#>   partyid                n
#>   <fct>              <int>
#> 1 Other                548
#> 2 Republican, strong  2314
#> 3 Republican, weak    3032
#> 4 Ind,near rep        1791
#> 5 Independent         4119
#> 6 Ind,near dem        2499
#> 7 Not str democrat    3690
#> 8 Strong democrat     3490
```

Simplifying Factor Levels with `fct_collapse()`

- `fct_collapse()` is ideal for grouping several levels together.

```
gss_cat |>
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know", "Other party"),
    rep = c("Strong republican", "Not str republican"),
    ind = c("Ind,near rep", "Independent", "Ind,near dem"),
    dem = c("Not str democrat", "Strong democrat")
  )) |>
  count(partyid)
#> # A tibble: 4 x 2
#>   partyid     n
#>   <fct>   <int>
#> 1 other     548
#> 2 rep      5346
#> 3 ind      8409
#> 4 dem      7180
```

Lumping Small Categories with `fct_lump()`

- `fct_lump()` groups less common levels into "Other" to simplify plots.

```
gss_cat |>
  mutate(relig = fct_lump(relig)) |>
  count(relig)
#> # A tibble: 2 x 2
#>   relig          n
#>   <fct>      <int>
#> 1 Protestant 10846
#> 2 Other      10637

# Or keep the top 3 most common levels
gss_cat |>
  mutate(relig = fct_lump(relig, n = 3)) |>
  count(relig, sort = TRUE)
#> # A tibble: 4 x 2
#>   relig          n
#>   <fct>      <int>
#> 1 Protestant 10846
#> 2 Catholic    5124
#> 3 None        3523
#> 4 Other       1990
```

Agenda

Basics of Strings in R

- **Creating Strings**: Use either double quotes `"` or single quotes `'` to define strings.

```
string1 <- "This is a string"
string2 <- 'To get a "quote" inside a string, use single quotes'
```

- **Escape Character**: The backslash `\` allows you to include special characters:
  - `\"` for double quotes, `\'` for single quotes, `\\` for a backslash.

```
double_quote <- "\""   # or '"'
single_quote <- '\''   # or "'"
```

String Display vs. Content

- The displayed representation of a string in R may include escape characters.

```
x <- c("\"", "\\")
x                # Displays escapes
#> [1] "\"" "\\"
writeLines(x)    # Raw contents without escapes
#> "
#> \
```

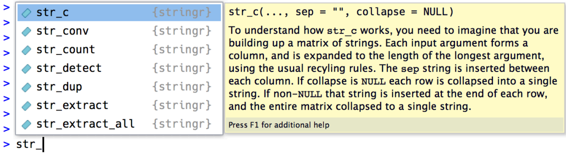
- Use `writeLines()` to view the actual contents of a string, bypassing escape characters in the display.

Special Characters and Useful String Functions

- **Special Characters**:
  - `"\n"` for newline, `"\t"` for tab.
  - Check additional characters using `?'"'` or `?"'"`.
- **Examples**:

```
(x <- "\u00b5")                   # Non-English character
#> [1] "µ"
c("one", "two", "three")          # Creating character vector
#> [1] "one"   "two"   "three"
str_length(c("a", "R for data science", NA)) # Calculate string length
#> [1]  1 18 NA
```

- **String Autocomplete**: `stringr` functions support autocomplete for easy access to string manipulation tools.



String Manipulation: Combining and Handling NA

- **Combining Strings**: Use `str_c()` to concatenate strings.

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

- **Handling Missing Values**: Use `str_replace_na()` to replace `NA` in strings.

```
x <- c("abc", NA)
str_c("|-", x, "-|")
#> [1] "|-abc-|" NA
str_c("|-", str_replace_na(x), "-|")
#> [1] "|-abc-|" "|-NA-|"
```

String Recycling and Collapsing

- **Recycling**: Extend strings to match vector lengths.

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
#> [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

- **Collapsing Vectors**: Collapse elements of a character vector into a single string.

```
str_c(c("x", "y", "z"), collapse = ", ")
#> [1] "x, y, z"
```

Subsetting strings

```r
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
#> [1] "App" "Ban" "Pea"
str_sub(x, -3, -1)
#> [1] "ple" "ana" "ear"
str_sub("a", 1, 5)
#> [1] "a"
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "apple"  "banana" "pear"
```

- See also `str_to_lower()`, `str_to_upper()`, or `str_to_title()`.

Locales and String Manipulation

- **Locales** affect how characters are treated based on language and regional rules.
  - Example: Turkish has two forms of the letter "i" — with and without a dot.
  - The capitalization behavior for these letters differs in Turkish.

```
# Default (system locale)
str_to_upper(c("i", "ı"))
#> [1] "I" "I"

# Turkish locale specified
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

- **Setting Locale**:
  - Use an ISO 639 language code, a two- or three-letter abbreviation.
  - If omitted, R uses the system's default locale.

Regular Expressions

"Some people, when confronted with a problem, think
'I know, I'll use regular expressions.'
Now, they have two problems."
— Jamie Zawinski

- **Regular expressions** (regex) are a powerful language for describing patterns in strings.
- **Regex Capabilities**:
  - **Match**: Identify strings that contain a specific pattern.
  - **Locate**: Find the exact positions where patterns occur in a string.
  - **Extract**: Retrieve the content that matches a pattern.
  - **Replace**: Substitute matches with a new value.
  - **Split**: Divide a string based on a pattern match.
- **Further Learning**:
  - Read the chapter on regex from *R for Data Science* for a comprehensive guide.

Basic Pattern Matching in Regex

- **View Pattern Matches**: Use `str_view()` to visualize pattern matches.
  - `str_view(string, pattern)` shows the pattern matches in the string.
- **Exact Match**: The simplest patterns match exact strings.

```
x <- c("apple", "banana", "pear")
str_view(x, "an")    # Matches "an" in the strings
#> [2] | b<an><an>a
```

- **Wildcard Character**: `.` matches any character (except newline).

```
str_view(x, ".a.")   # Matches any character, "a", and another character
#> [2] | <ban>ana
#> [3] | p<ear>
```

Escaping Special Characters

- Since `.` matches any character, how do we match a literal `.`?
  - **Escape** it with `\\.`: backslash (\) is the escape character.

```
# Define regex for matching a literal dot
dot <- "\\."
writeLines(dot)      # Actual content is a single dot
#> \.
str_view(c("abc", "a.c", "bef"), "a\\.c")  # Matches "a.c" only
#> [2] | <a.c>
```

Matching Literal Backslashes

- To match a literal \, use \\ as the regex.
  - **Double the escape**: to represent \\, use \\\\ in the string.

```
x <- "a\\b"
writeLines(x)          # Shows "a\b"
#> a\b
str_view(x, "\\\\")   # Matches a single literal backslash
#> [1] | a<\>b
```

Anchors in Regex

- By default, regex patterns match any part of a string.
- **Anchors** help specify where in the string to match:
  - ^ for the **start** of a string.
  - $ for the **end** of a string.

```
x <- c("apple", "banana", "pear")
```

- **Example: Start Anchor**

```
# Matches "a" at the start
str_view(x, "^a")
#> [1] | <a>pple
```

- **Example: End Anchor**

```
# Matches "a" at the end
str_view(x, "a$")
#> [2] | banan<a>
```

Matching the Entire String

- To match a complete string, use both ^ and $:

```r
x <- c("apple pie", "apple", "apple cake")
```

- **Partial Match**

```r
# Matches anywhere
str_view(x, "apple")
#> [1] | <apple> pie
#> [2] | <apple>
#> [3] | <apple> cake
```

- **Full String Match**

```r
# Matches as the entire string
str_view(x, "^apple$")
#> [2] | <apple>
```

Character Classes and Alternatives

- **Special Patterns**:
  - `.`: Matches any character except newline.
  - `\\d`: Matches any digit (remember to escape \).
  - `\\s`: Matches any whitespace character.
- **Character Classes**:
  - `[abc]`: Matches `a`, `b`, or `c`.
  - `[^abc]`: Matches anything except `a`, `b`, or `c`.
- **Alternatives**:
  - Use | to match multiple patterns.
  - Example: `abc|def` matches `"abc"` or `"def"`.

Character Classes in Practice

- Character classes provide flexibility in matching.

- **Exact Character Match**

```
str_view(
  c(
    "abc",
    "a.c",
    "a*c",
    "a c"
  ),
  "a[.]c"
)
#> [2] | <a.c>
```

- **Match a Specific Character**

```
str_view(
  c(
    "abc",
    "a.c",
    "a*c",
    "a c"
  ),
  ".[*]c"
)
#> [3] | <a*c>
```

- **Match Spaces Explicitly**

```
str_view(
  c(
    "abc",
    "a.c",
    "a*c",
    "a c"
    ),
  "a[ ]"
  )
#> [4] | <a >c
```

Using Alternatives

- **Alternatives** with `|` allow pattern flexibility.
  - Remember, `|` has low precedence.
  - `gr(e|a)y`: Matches both "grey" and "gray".

```
str_view(c("grey", "gray"), "gr(e|a)y")
#> [1] | <grey>
#> [2] | <gray>
```

Repetition in Regex

- Control the number of matches with these quantifiers:
  - ?: Matches 0 or 1 times.
  - +: Matches 1 or more times.
  - *: Matches 0 or more times.

```
# Example string: Longest year in Roman numerals
x <- "MDCCCLXXXVIII"
```

- **Optional Match**

```
str_view(x, "CC?")
#> [1] | MD<CC><C>LXXXVIII
```

- **One or More Matches**

```
str_view(x, "CC+")
#> [1] | MD<CCC>LXXXVIII
```

- **Match Group**

```
str_view(x, 'C[LX]+')
#> [1] | MDCC<CLXXX>VIII
```

Specifying Exact Match Counts

- **Precise Quantifiers**:
  - `{n}`: Exactly `n` matches.
  - `{n,}`: `n` or more matches.
  - `{n,m}`: Between `n` and `m` matches.

- **Match Exactly 2**

```
str_view(x, "C{2}")
#> [1] | MD<CC>CLXXXVIII
```

- **Match 2 or More**

```
str_view(x, "C{2,}")
#> [1] | MD<CCC>LXXXVIII
```

- **Match Between 2 and 3**

```
str_view(x, "C{2,3}")
#> [1] | MD<CCC>LXXXVIII
```

Grouping and Backreferences

- Parentheses (`()`) in regex:
  - Organize complex patterns.
  - **Capture** parts of a match, storing them as **numbered groups**.
- **Backreferences**:
  - Refer back to previously captured groups using \1, \2, etc.

```
str_view(stringr::fruit, "(..)\\1")
#>  [4] | b<anan>a
#> [20] | <coco>nut
#> [22] | <cucu>mber
#> [41] | <juju>be
#> [56] | <papa>ya
#> [73] | s<alal> berry
```

- Imagine as `(..)` capturing the first two characters, and `\\1` referencing the first captured group.
- This example finds words with repeated pairs of letters, like "banana" (`(an)(an)`) or "cucumber" (`(cu)(cu)`).

Grouping and Backreferences II

- More advanced patterns with backreferences:
  - Example: Finding words that start and end with the same two letters:

```
str_view(stringr::words, "^(..).*\\1$")
#> [152] | <church>
#> [217] | <decide>
#> [617] | <photograph>
#> [699] | <require>
#> [739] | <sense>
```

- **Reordering Using Capturing Groups**:
  - Use `str_replace()` with backreferences to reorder text, referencing each captured group.

```
stringr::sentences |>
  str_replace("(\\w+) (\\w+) (\\w+)", "\\1 \\3 \\2") |>
  head(2)
#> [1] "The canoe birch slid on the smooth planks."
#> [2] "Glue sheet the to the dark blue background."
```

This example reorders the second and third words in sentences.

Non-Capturing Groups

- **Non-capturing groups**:
  - Useful when you want to group patterns without capturing them, preventing unnecessary backreference groups.
  - Use `(?:...)` for non-capturing.

```
x <- c("a gray cat", "a grey dog")
str_match(x, "gr(?:e|a)y")
#>      [,1]
#> [1,] "gray"
#> [2,] "grey"
```

- **Example**:
  - `(?:e|a)` specifies either "e" or "a" without creating an extra backreference group.

This can simplify patterns by omitting captures where they aren't needed, while still using grouping to clarify complex regex structures.