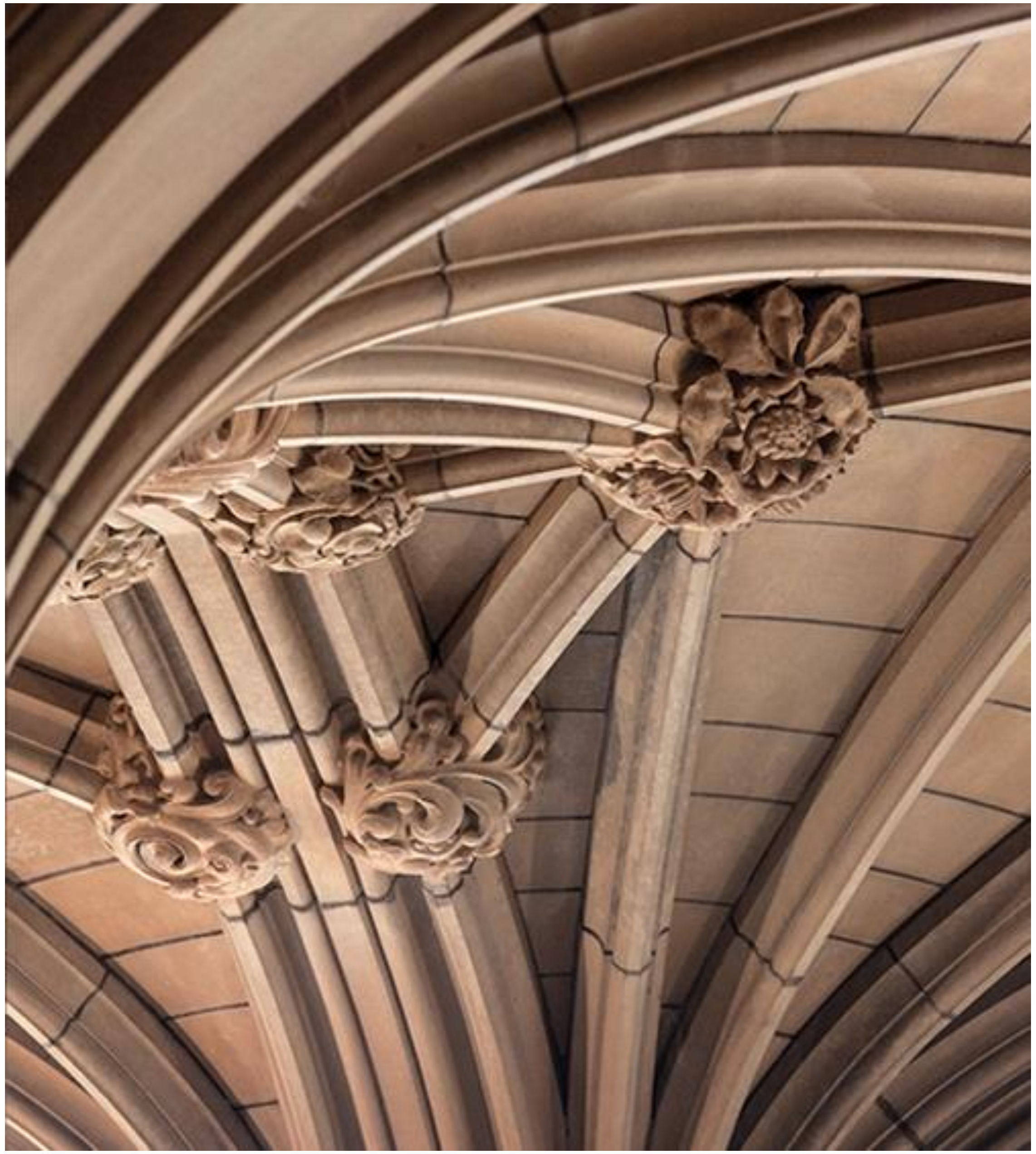
Catchment areas, kernel densities and Inverse Distance Weighting in R

Dr Adrian Ellison
Dr Richard Ellison

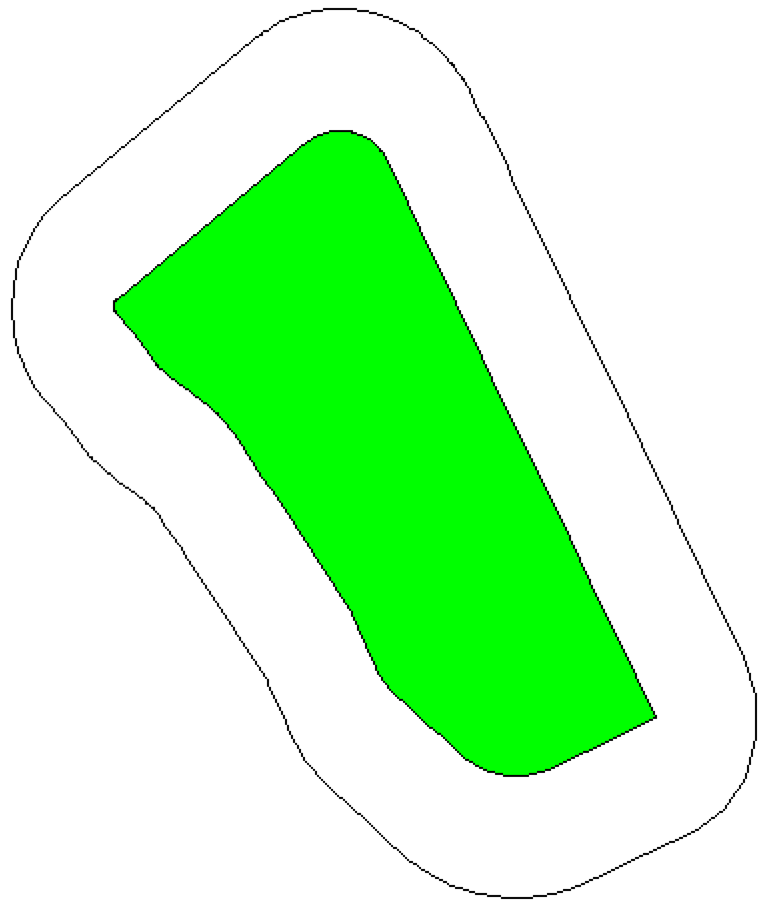
ITLS6107 Semester 2, 2017

THE UNIVERSITY OF SYDNEY



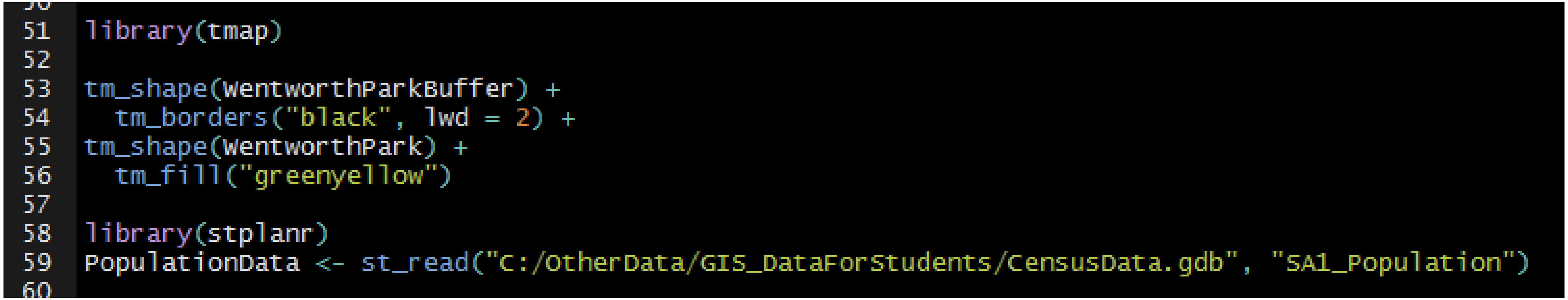
Simple buffers in R

- What if we wanted to do more than simply plot the area?
- If we needed to calculate the population within a distance of a location we can use several functions together.
- Alternatively, we can use the stplanr package's calc_catchment function together with our population data



Catchment areas in R

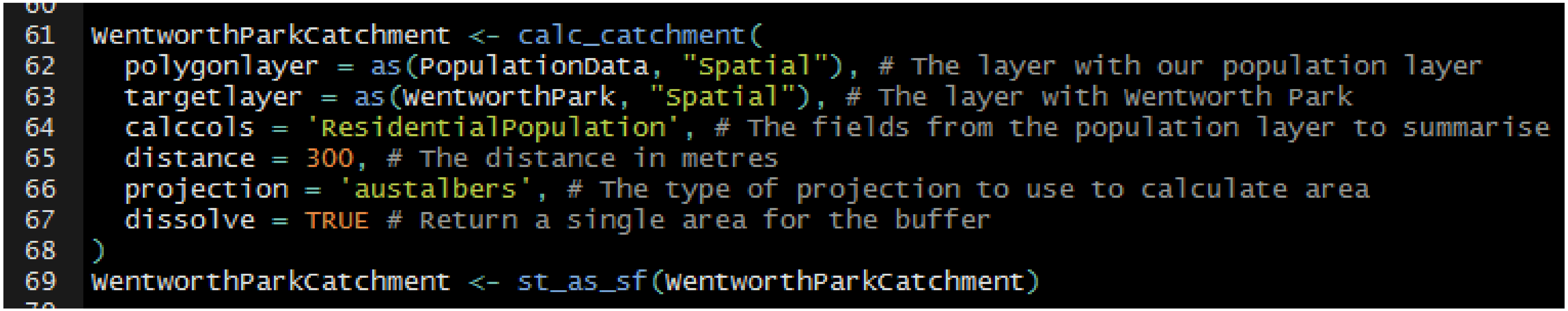
– We first need to install and load the stplanr package
– We also need to import our population layer



```
library(tmap)

tm_shape(WentworthParkBuffer) +
  tm_borders("black", lwd = 2) +
tm_shape(WentworthPark) +
  tm_fill("greenyellow")

library(stplanr)
PopulationData <- st_read("C:/OtherData/GIS_DataForStudents/CensusData.gdb", "SA1_Population")
```

Catchment areas in R

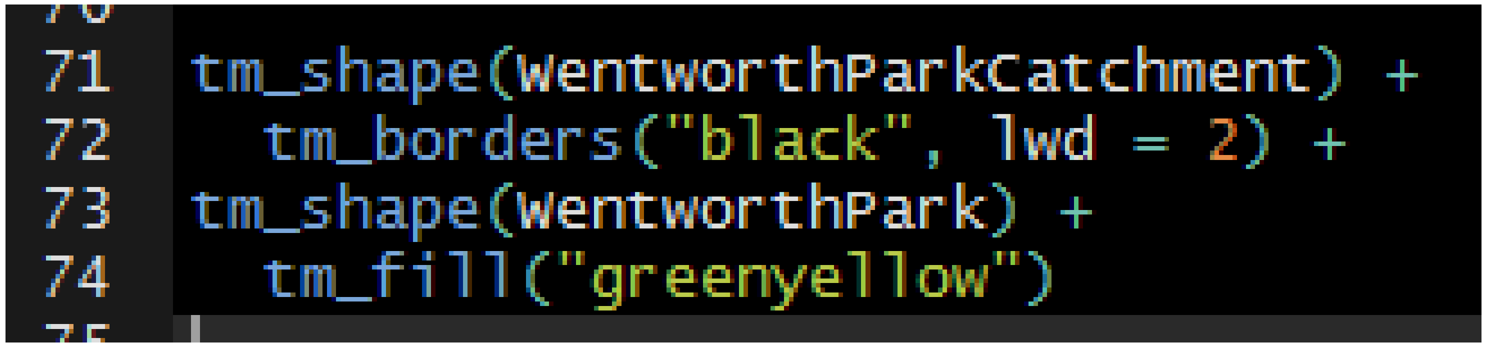
- The calc_catchment function takes the population layer, a layer with the location (points, lines or polygons), the distance and some other options.
- The current version requires **sp** layers so we will convert them
- We can convert the result back to an **sf** layer



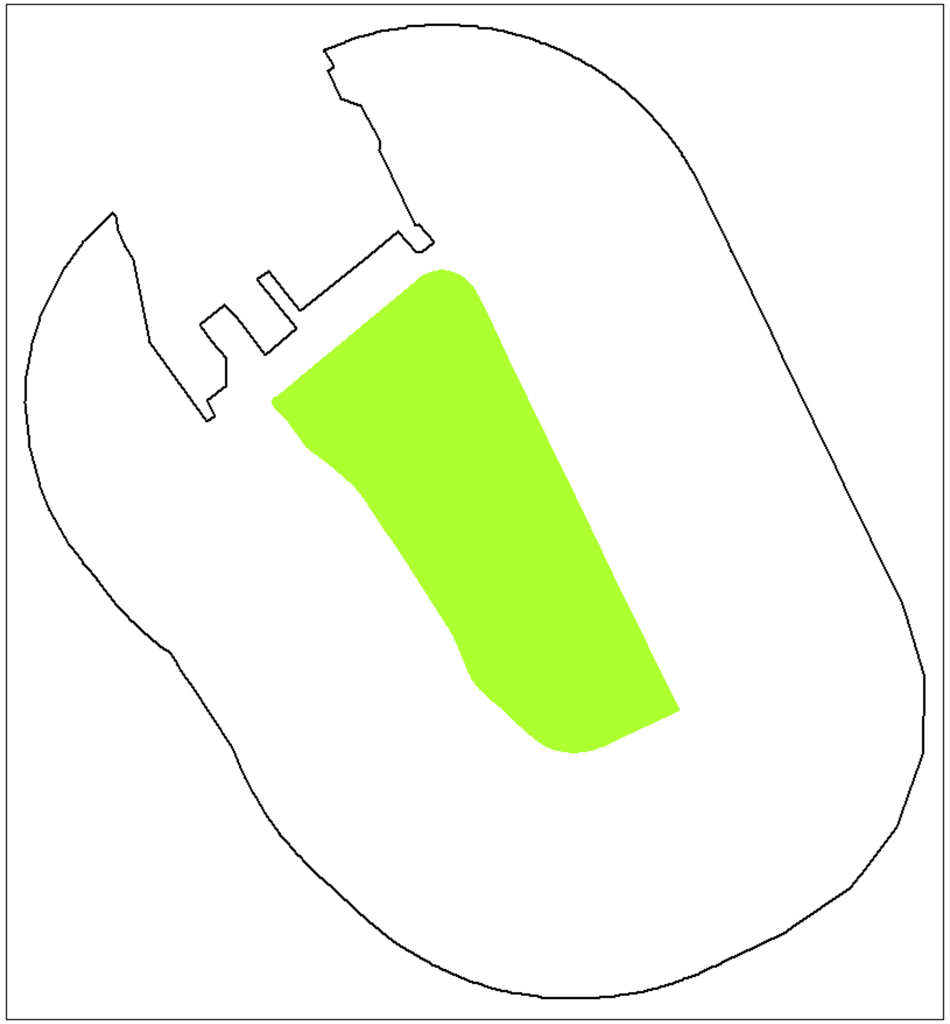
```
WentworthParkCatchment <- calc_catchment(
  polygonlayer = as(PopulationData, "Spatial"), # The layer with our population layer
  targetlayer = as(WentworthPark, "Spatial"), # The layer with Wentworth Park
  calccols = 'ResidentialPopulation', # The fields from the population layer to summarise
  distance = 300, # The distance in metres
  projection = 'austalbers', # The type of projection to use to calculate area
  dissolve = TRUE # Return a single area for the buffer
)
WentworthParkCatchment <- st_as_sf(WentworthParkCatchment)
```

Catchment areas in R

– The result can be plotted just like our previous buffer

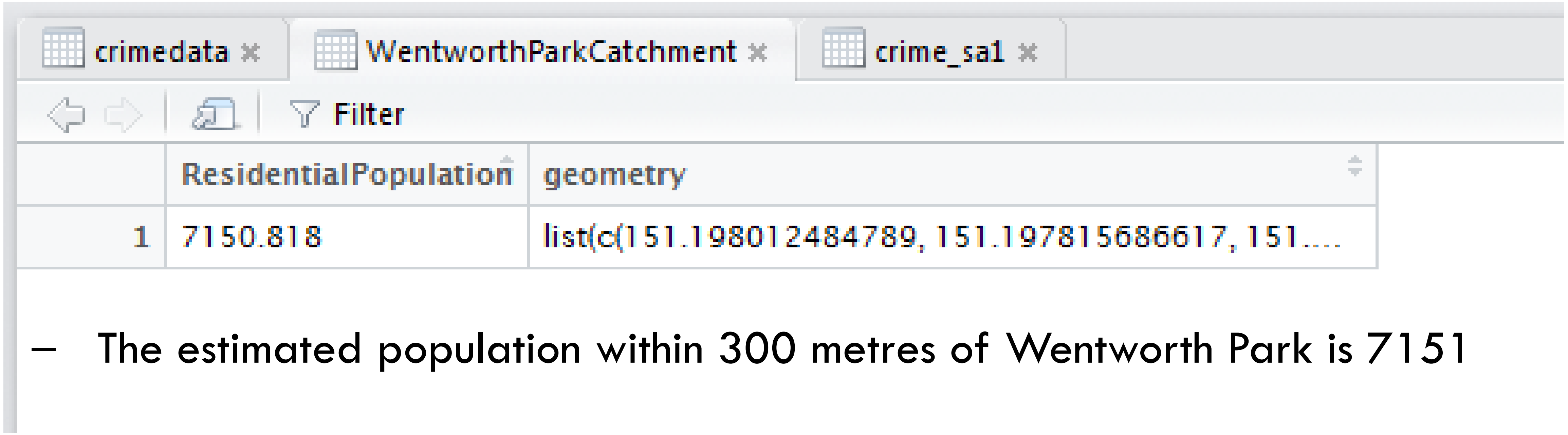


```
tm_shape(WentworthParkCatchment) +
  tm_borders("black", lwd = 2) +
tm_shape(WentworthPark) +
  tm_fill("greenyellow")
```



Catchment areas in R

– We can use View(WentworthParkCatchment) to look at the summary statistics

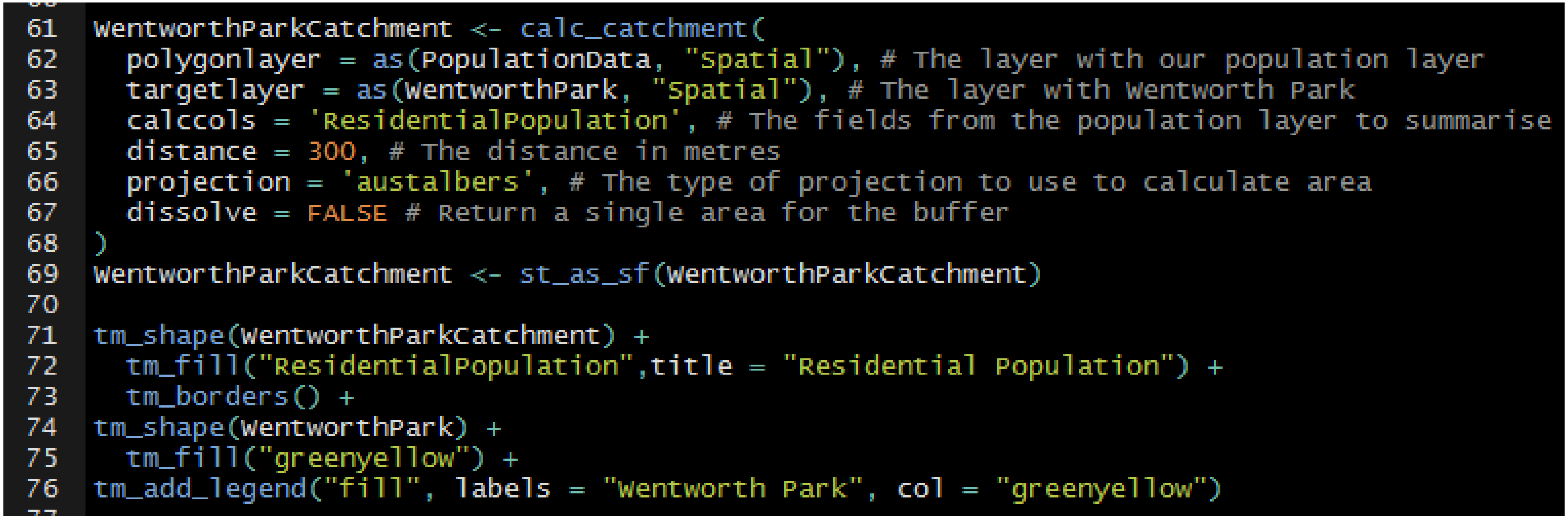


	ResidentialPopulation	geometry
1	7150.818	list(c(151.198012484789, 151.197815686617, 151....

– The estimated population within 300 metres of Wentworth Park is 7151

Catchment areas in R

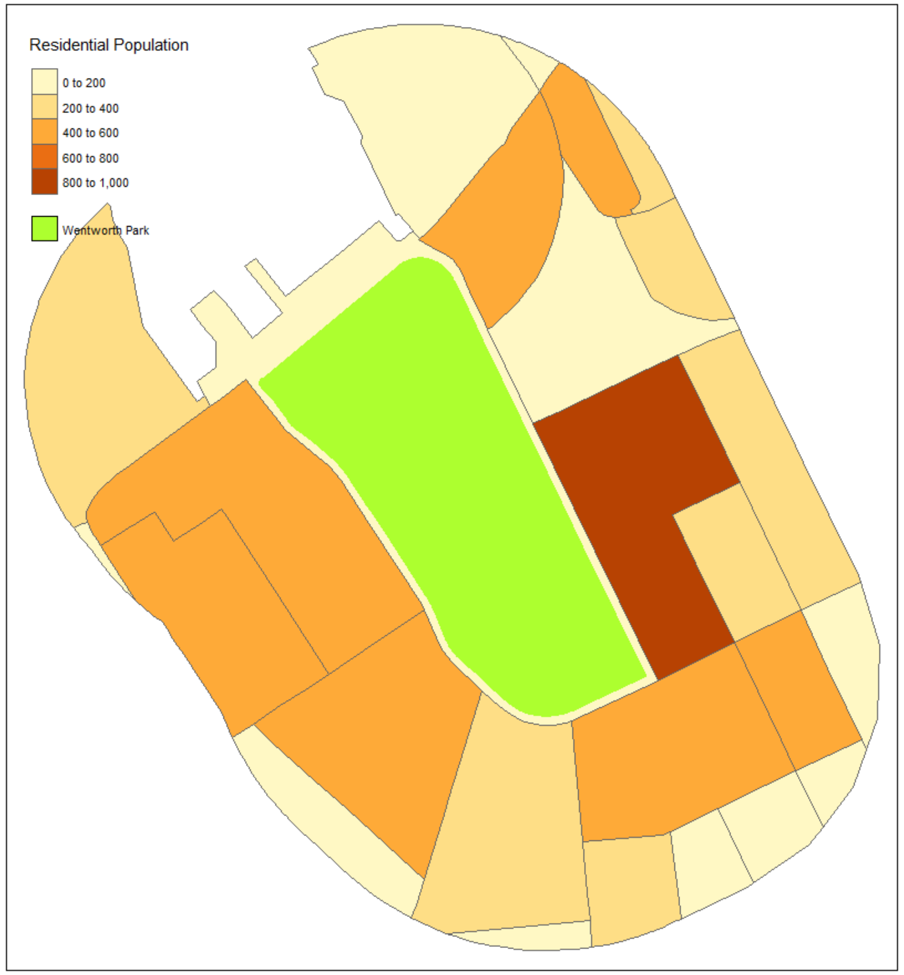
– If we choose dissolve = FALSE then we can see the population estimates for each SA1 that falls within the catchment area:



```r
WentworthParkCatchment <- calc_catchment(
  polygonlayer = as(PopulationData, "Spatial"), # The layer with our population layer
  targetlayer = as(WentworthPark, "Spatial"), # The layer with Wentworth Park
  calccols = 'ResidentialPopulation', # The fields from the population layer to summarise
  distance = 300, # The distance in metres
  projection = 'austalbers', # The type of projection to use to calculate area
  dissolve = FALSE # Return a single area for the buffer
)
WentworthParkCatchment <- st_as_sf(WentworthParkCatchment)

tm_shape(WentworthParkCatchment) +
  tm_fill("ResidentialPopulation",title = "Residential Population") +
  tm_borders() +
tm_shape(WentworthPark) +
  tm_fill("greenyellow") +
tm_add_legend("fill", labels = "Wentworth Park", col = "greenyellow")
```

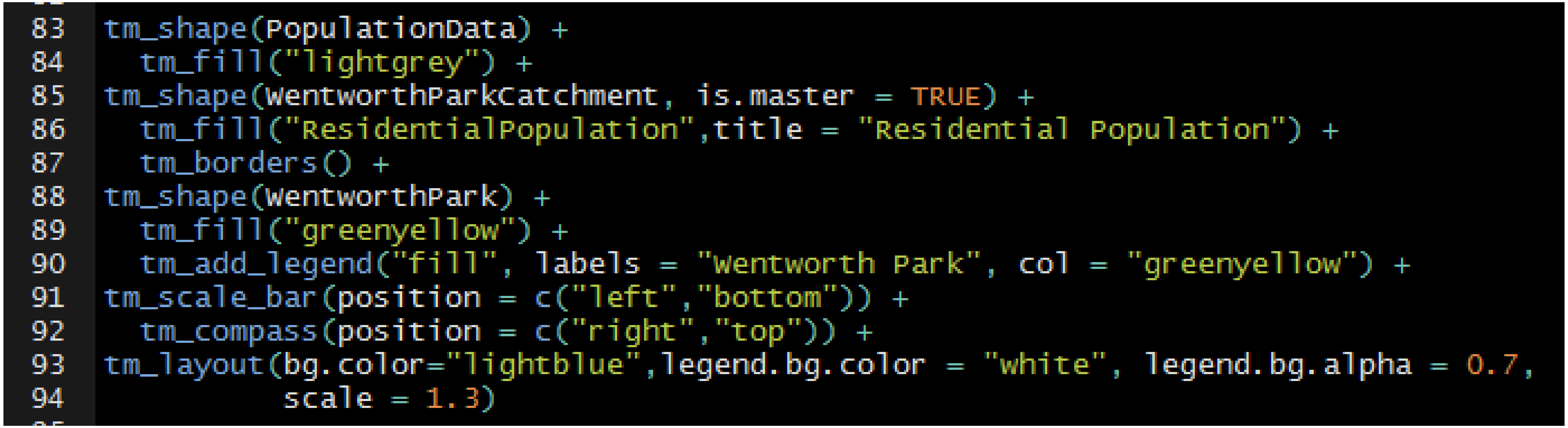
Catchment areas in R

- If we choose dissolve = FALSE then we can see the population estimates for each SA1 that falls within the catchment area:



Catchment areas in R

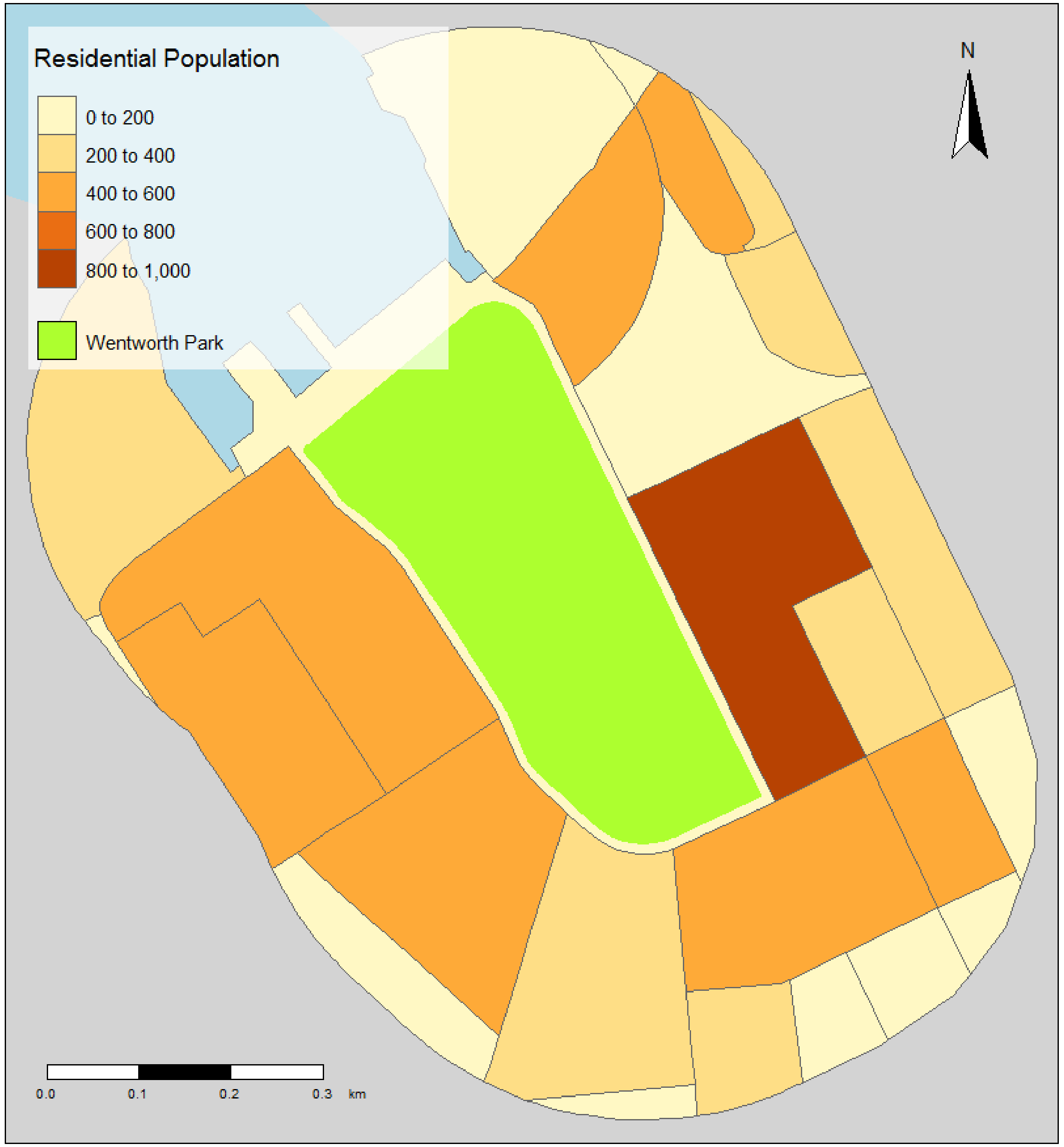
– We can add the original layer and make other layout changes:



```
tm_shape(PopulationData) +
  tm_fill("lightgrey") +
tm_shape(WentworthParkCatchment, is.master = TRUE) +
  tm_fill("ResidentialPopulation",title = "Residential Population") +
  tm_borders() +
tm_shape(WentworthPark) +
  tm_fill("greenyellow") +
  tm_add_legend("fill", labels = "Wentworth Park", col = "greenyellow") +
tm_scale_bar(position = c("left","bottom")) +
  tm_compass(position = c("right","top")) +
tm_layout(bg.color="lightblue",legend.bg.color = "white", legend.bg.alpha = 0.7,
          scale = 1.3)
```

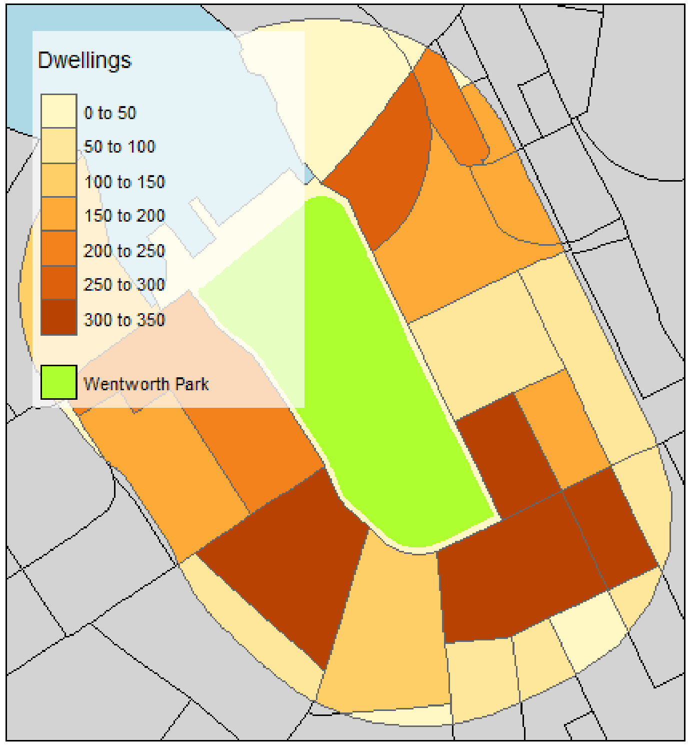
Catchment areas in R

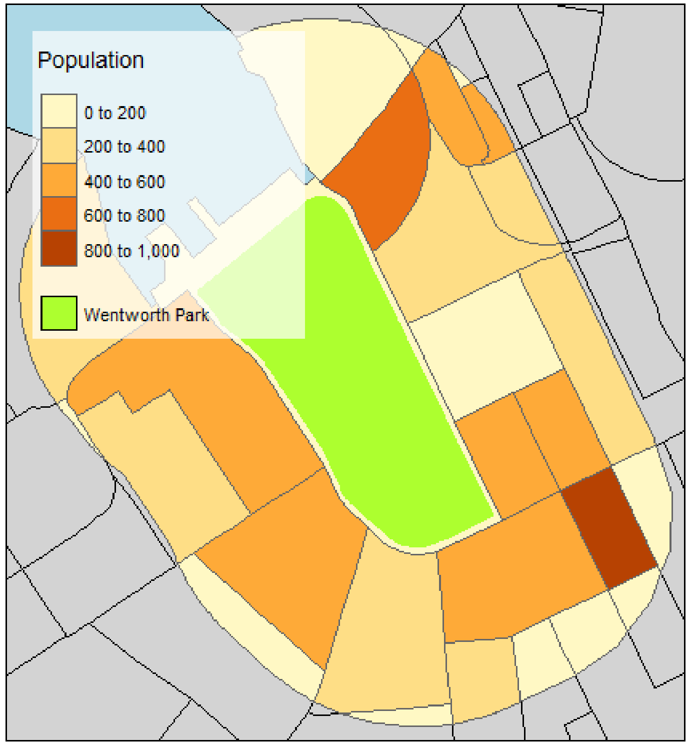
- We can add the original layer and make other layout changes



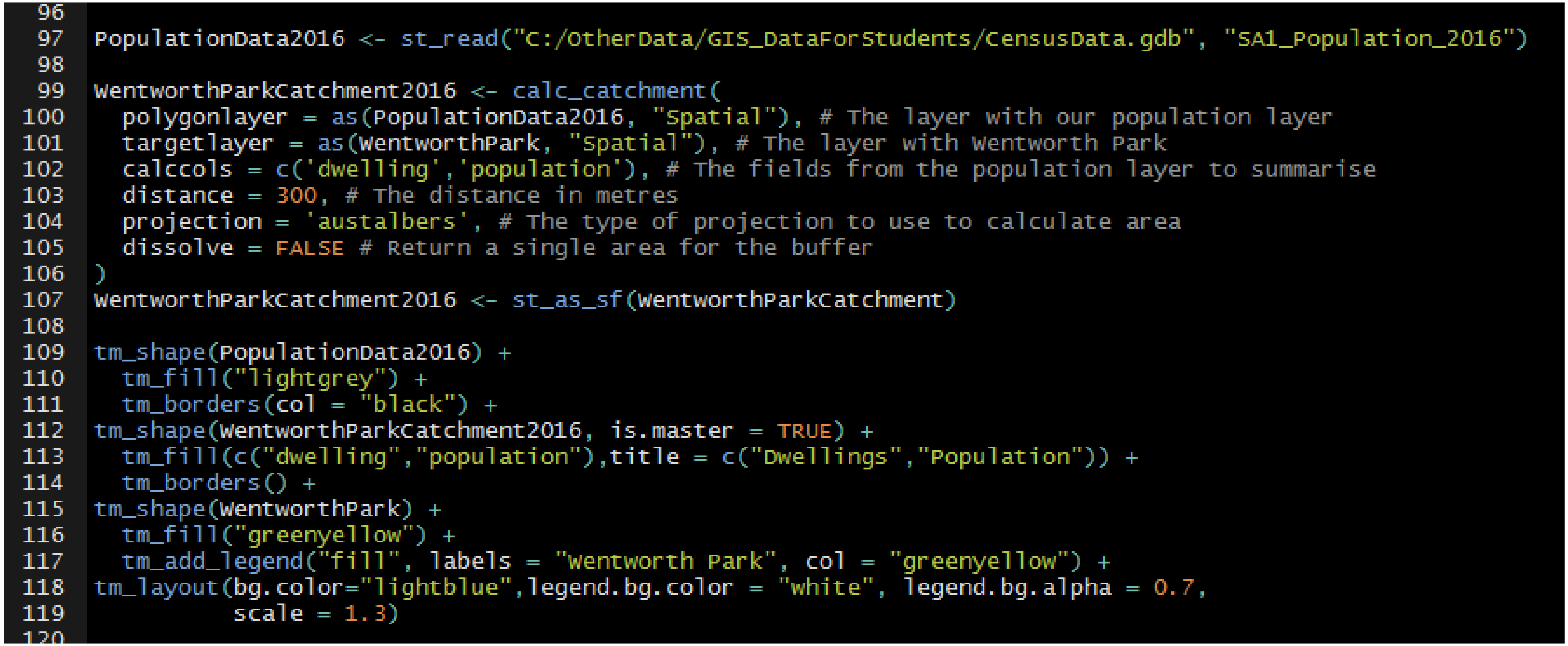
Small multiples and facets with tmap

- Sometimes you want to plot different variables on multiple small maps instead of one large map.
- The tmap package provides a way to easy display multiple variables as different themes on adjacent maps:





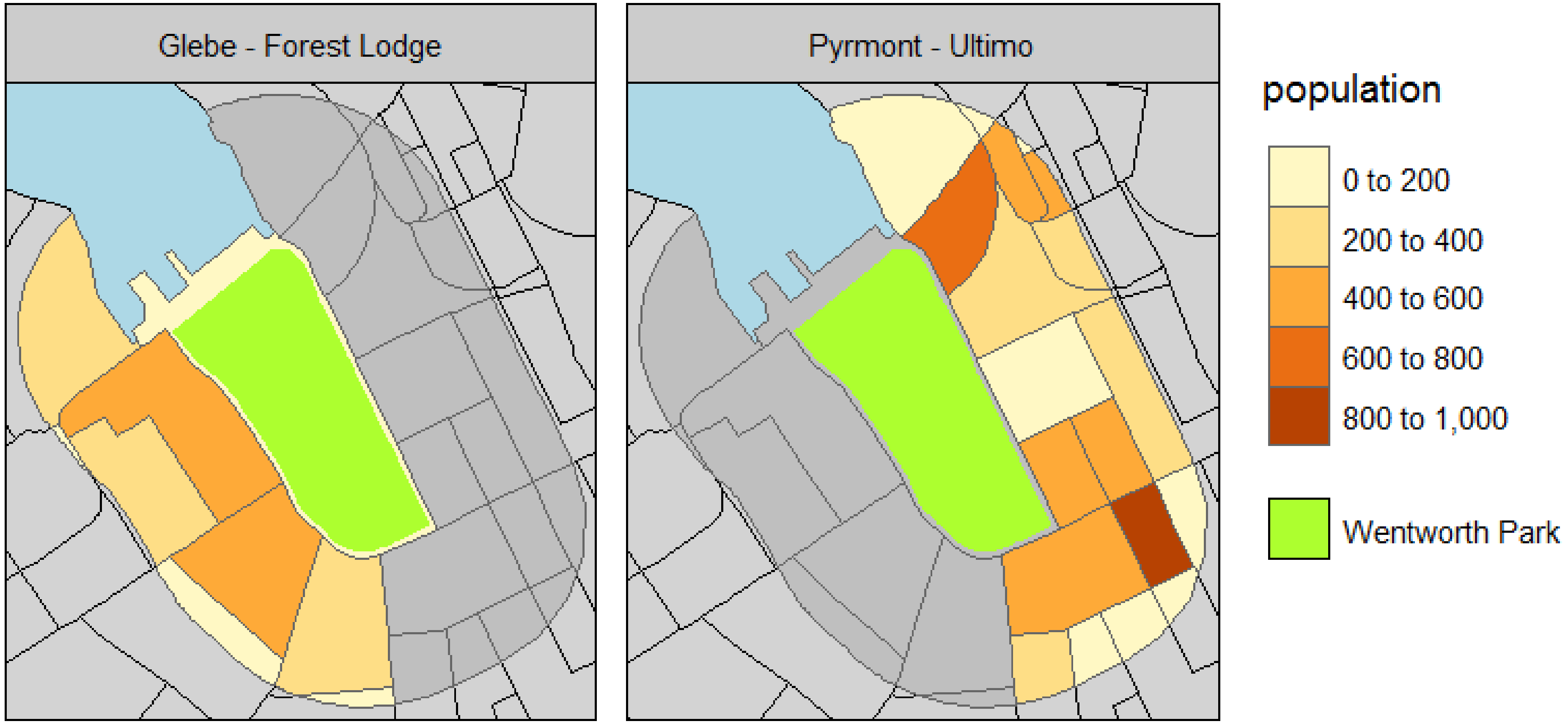
Small multiples and facets with tmap



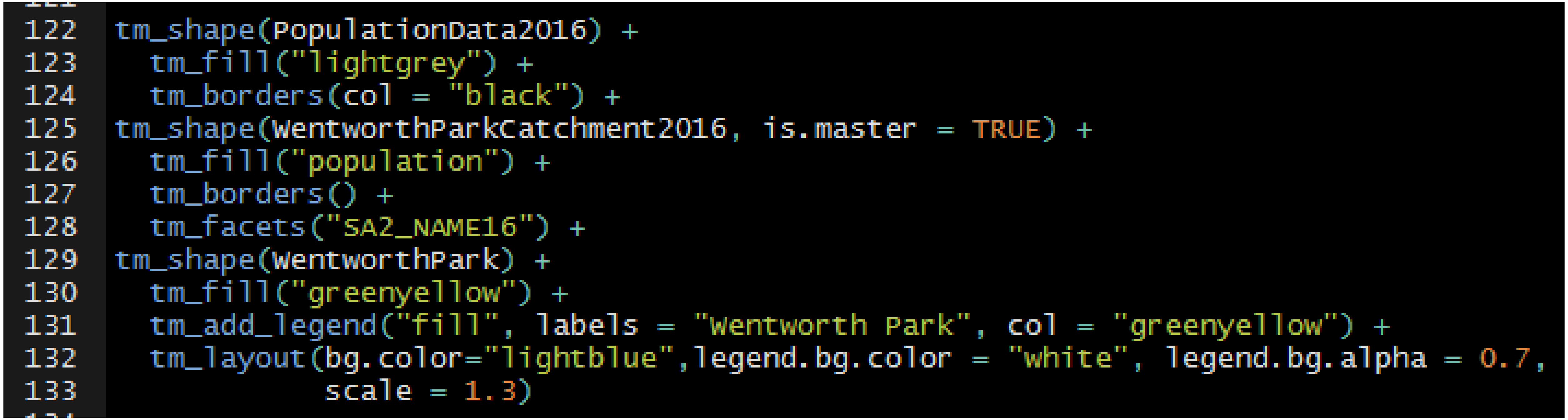
```

PopulationData2016 <- st_read("C:/OtherData/GIS_DataForStudents/CensusData.gdb", "SA1_Population_2016")

WentworthParkCatchment2016 <- calc_catchment(
  polygonlayer = as(PopulationData2016, "Spatial"), # The layer with our population layer
  targetlayer = as(WentworthPark, "Spatial"), # The layer with Wentworth Park
  calccols = c('dwelling','population'), # The fields from the population layer to summarise
  distance = 300, # The distance in metres
  projection = 'austalbers', # The type of projection to use to calculate area
  dissolve = FALSE # Return a single area for the buffer
)
WentworthParkCatchment2016 <- st_as_sf(WentworthParkCatchment)

tm_shape(PopulationData2016) +
  tm_fill("lightgrey") +
  tm_borders(col = "black") +
tm_shape(WentworthParkCatchment2016, is.master = TRUE) +
  tm_fill(c("dwelling","population"),title = c("Dwellings","Population")) +
  tm_borders() +
tm_shape(WentworthPark) +
  tm_fill("greenyellow") +
  tm_add_legend("fill", labels = "Wentworth Park", col = "greenyellow") +
tm_layout(bg.color="lightblue",legend.bg.color = "white", legend.bg.alpha = 0.7,
          scale = 1.3)

```

Small multiples and facets with tmap

- Sometimes we want different features/rows to be displayed separately rather than different variables
- We use the tm_facets() function to split our rows into groups for display



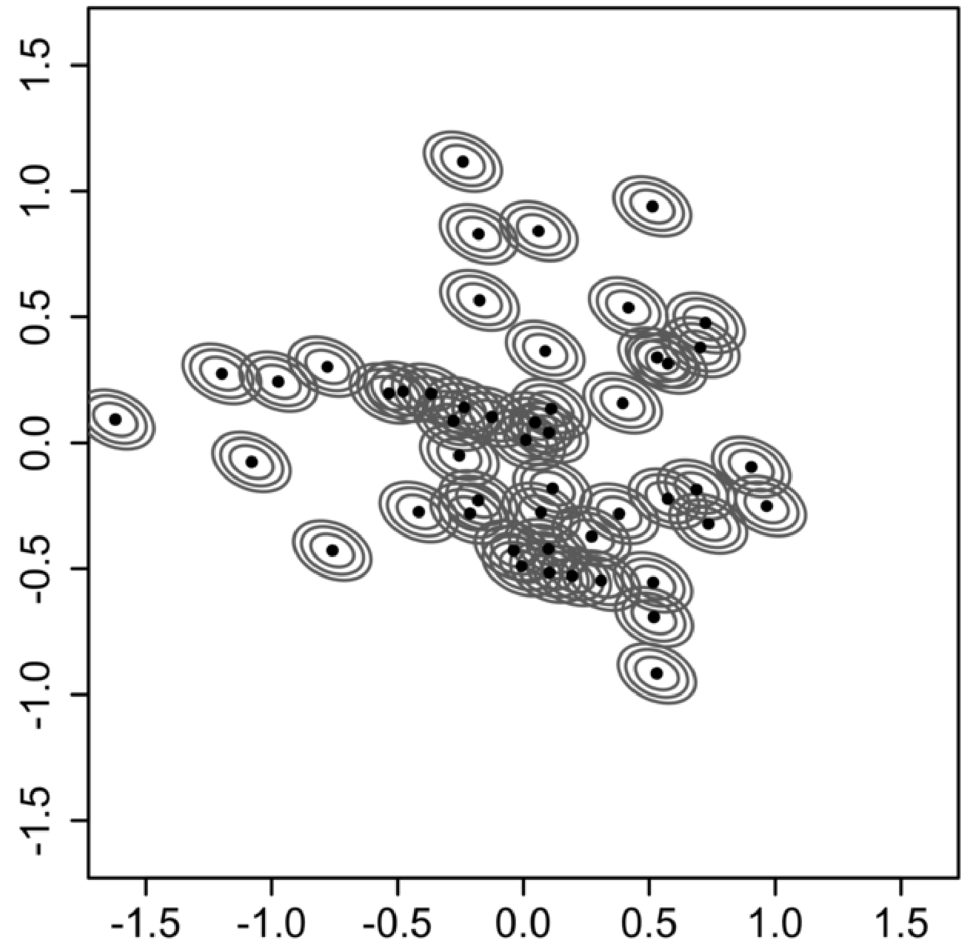
Small multiples and facets with tmap

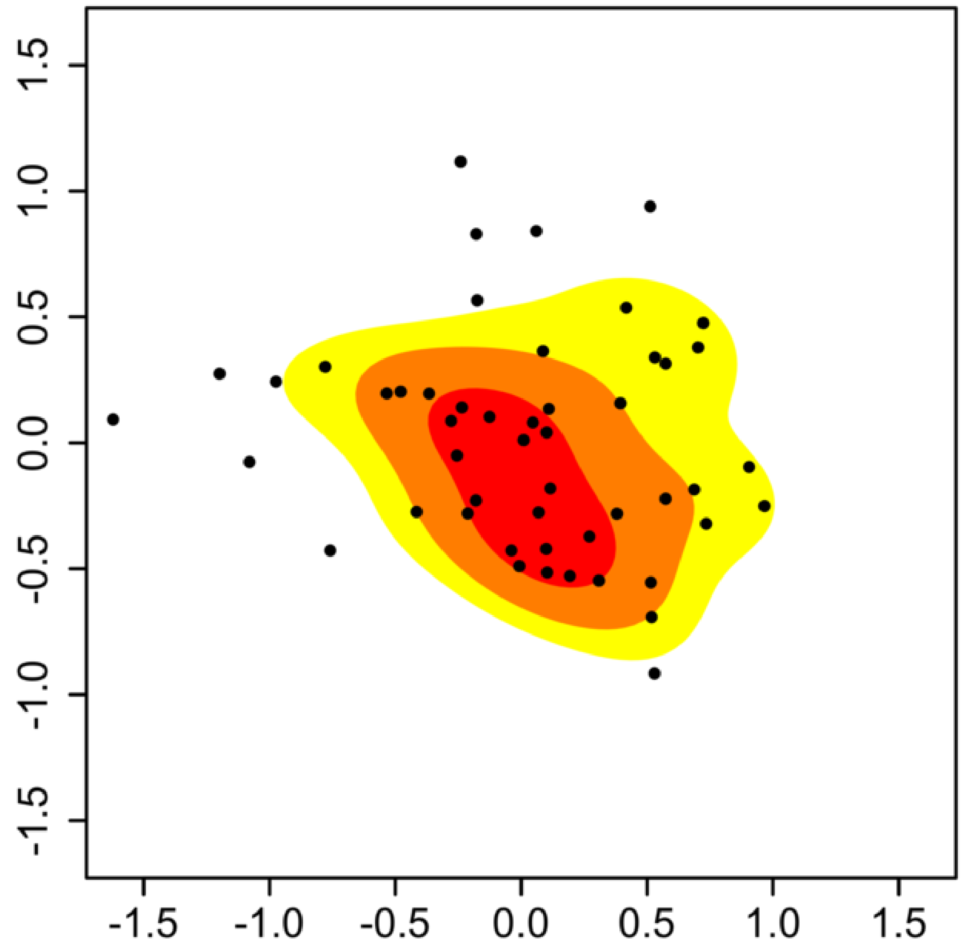


```
tm_shape(PopulationData2016) +
  tm_fill("lightgrey") +
  tm_borders(col = "black") +
tm_shape(WentworthParkCatchment2016, is.master = TRUE) +
  tm_fill("population") +
  tm_borders() +
  tm_facets("SA2_NAME16") +
tm_shape(WentworthPark) +
  tm_fill("greenyellow") +
  tm_add_legend("fill", labels = "Wentworth Park", col = "greenyellow") +
  tm_layout(bg.color="lightblue",legend.bg.color = "white", legend.bg.alpha = 0.7,
            scale = 1.3)
```

What is spatial kernel density estimation?

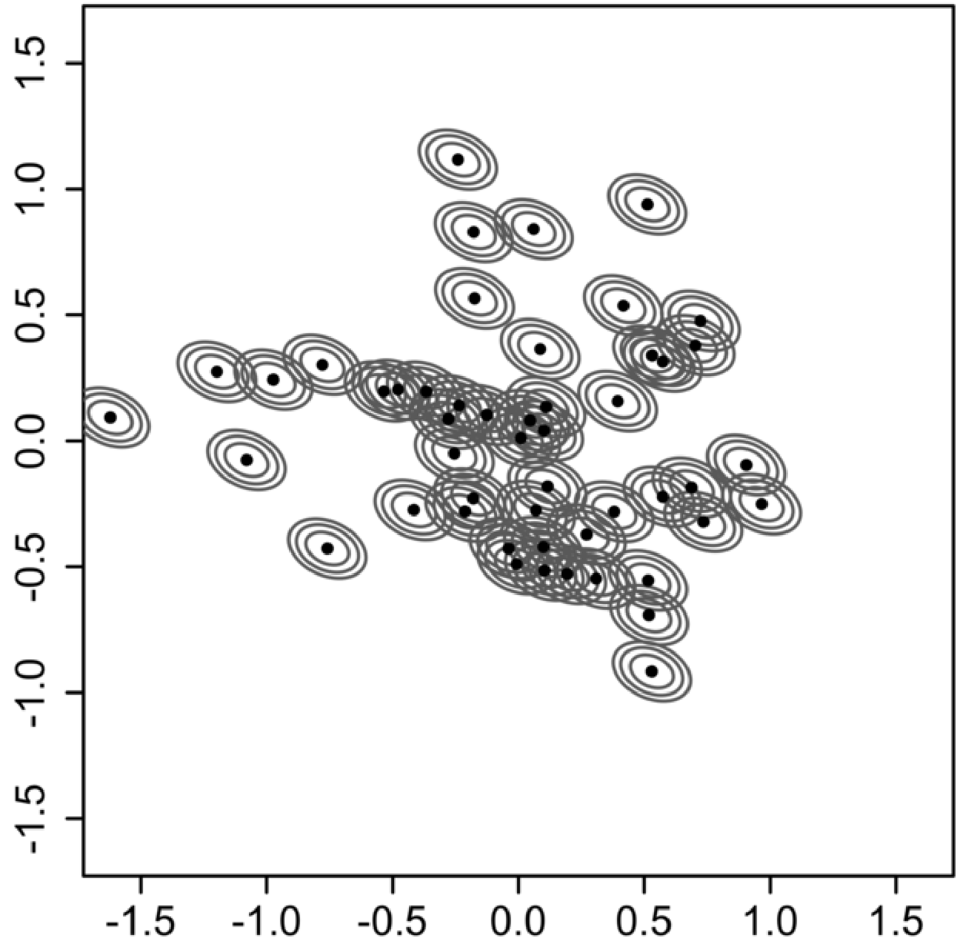
– Computes a distribution of values along two dimensional space using small "buffers"

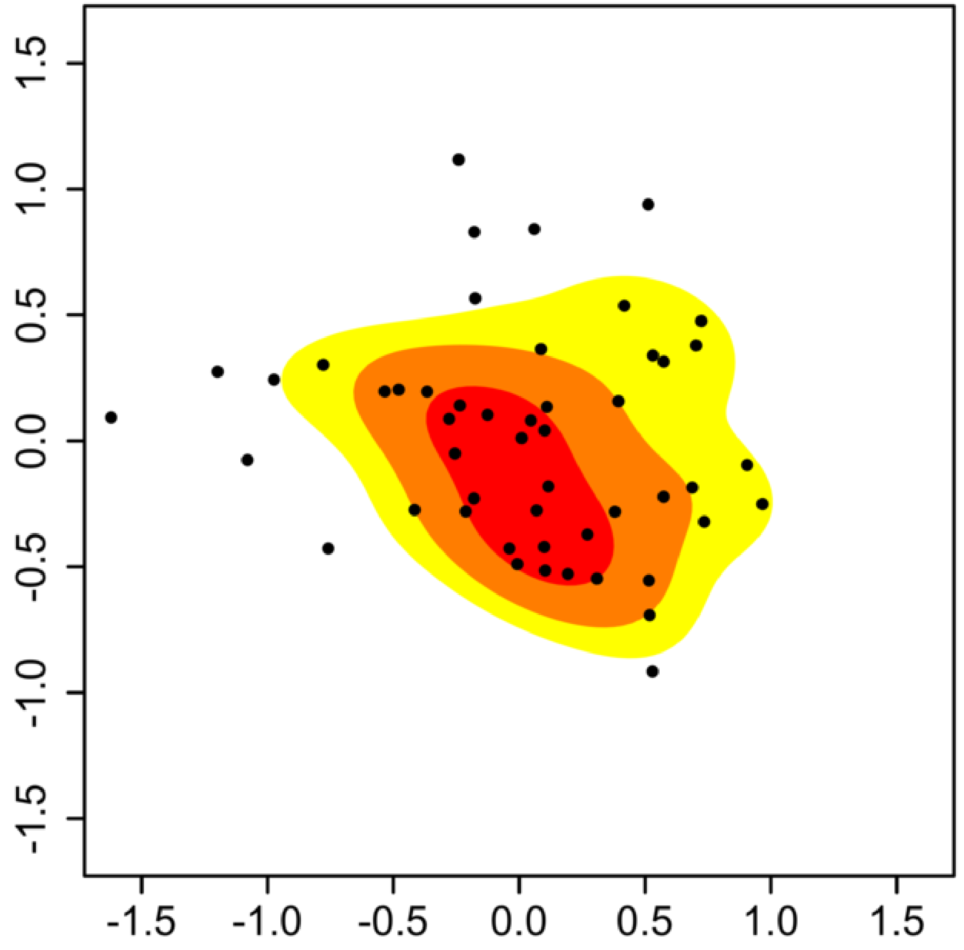




What is spatial kernel density estimation?

- Allows us to calculate the spatial density and distribution of point data





Kernel density estimation in R

- Many possible methods of calculating kernel densities in R using functions from different packages
- In almost all cases, requires a projected layer
  - Means we must project our crimelocs layer:

```
crimelocs <- spTransform(crimelocs, CRS(get_proj4(3308)))
```

  - We are converting to the NSW Lambert projection
  - 3308 is a standard code (EPSG) for this projection
    - 4326 is the code for Unprojected WGS 1984

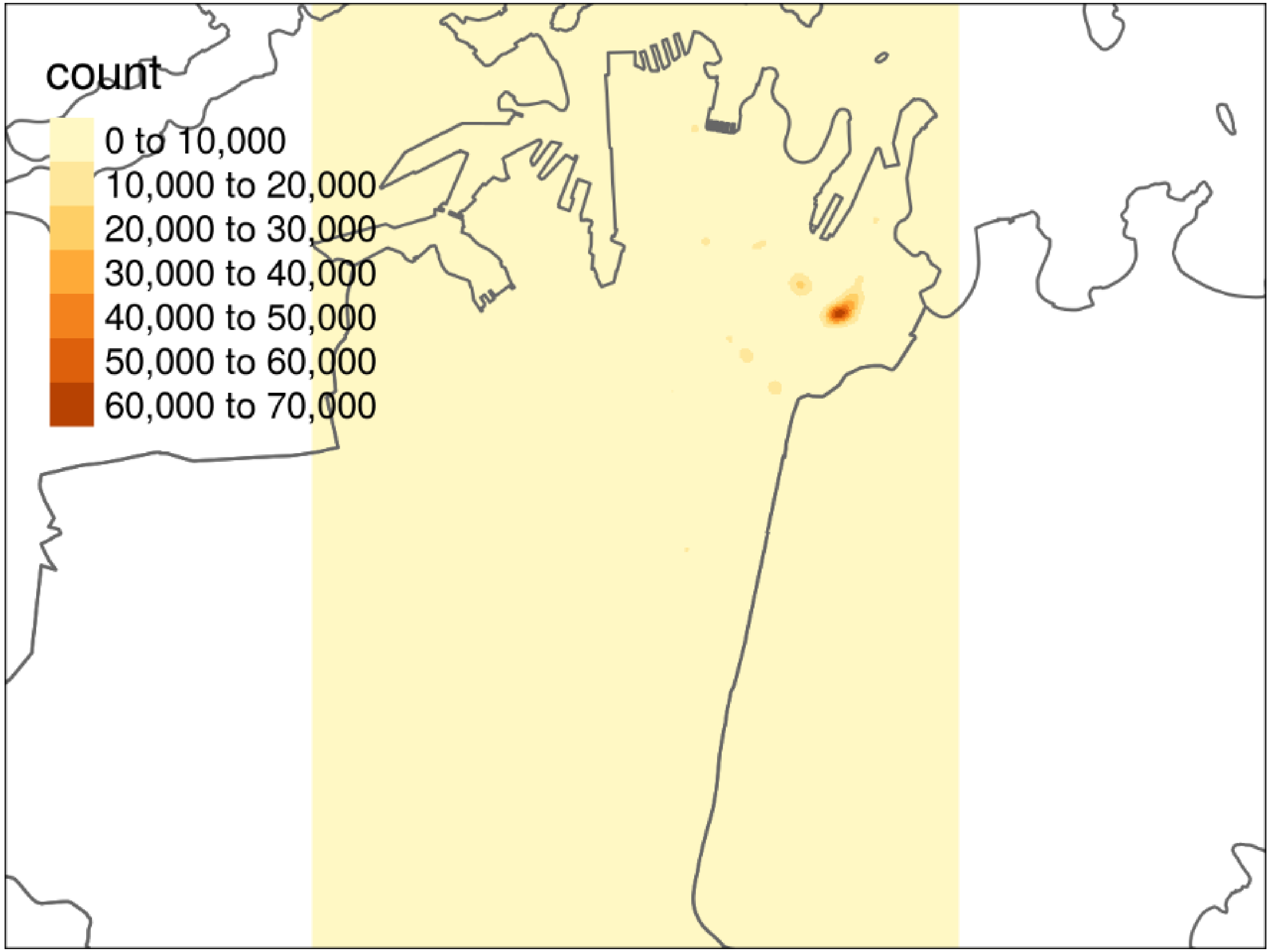
Kernel density estimation in R

- The **tmaptools** package contains a spatial kernel density function called **smooth_map**

```
crimekd <- smooth_map(crimelocs)
tm_shape(crimekd$raster) + tm_raster(col="count")
```

- The smooth_map function computes the kernel density and returns the results in several formats:
  - Raster
  - Line (similar to contour lines or isobars)
  - Polygons
- We can plot the result using tmap

Kernel density estimation in R



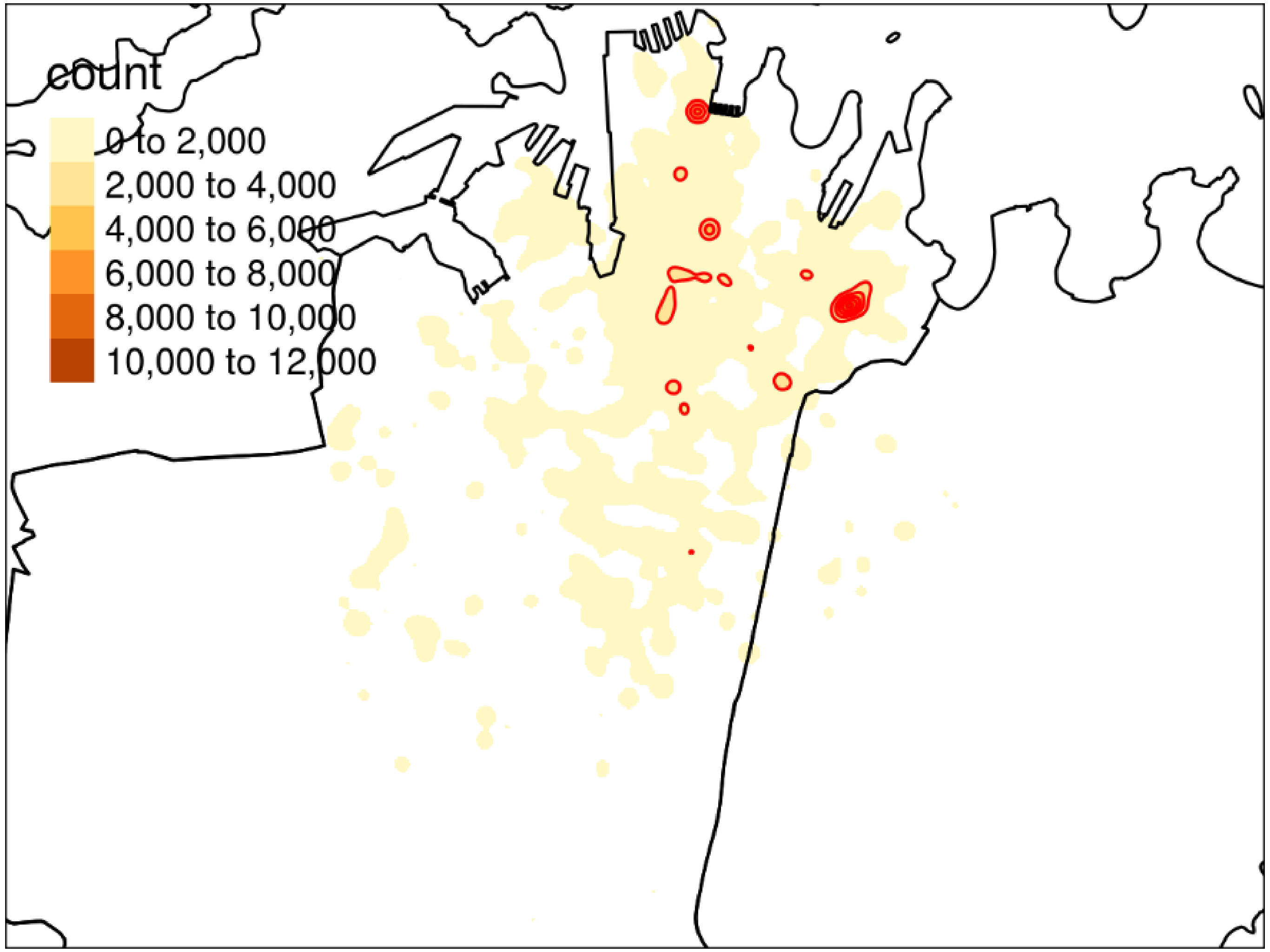
Kernel density estimation in R

– We can adjust the map to make it a bit clearer:

```r
sydsa4 <- readOGR("E:/data/CensusData.gdb","Sydney_SA4")
sydsa4 <- spTransform(sydsa4, CRS(get_proj4(3308)))

tm_shape(crimekd$raster[which(crimekd$raster$count > 100),]) +
       tm_raster(col="count", alpha=0.75) +
tm_shape(crimekd$iso) +
       tm_lines() +
tm_shape(sydsa4) +
       tm_borders(col="black")
```

Kernel density estimation in R

– Kernel density for assaults only:



Comments on kernel density estimation

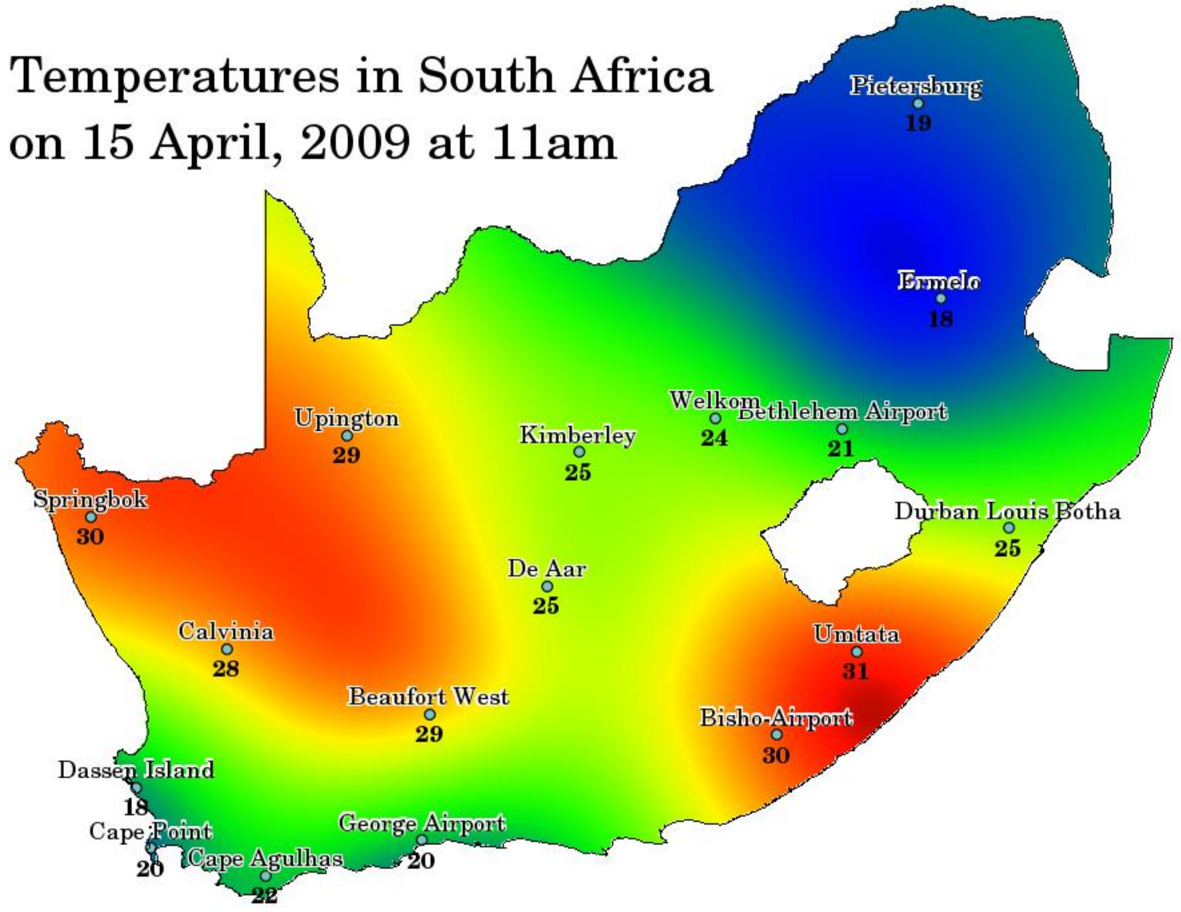
- Kernel density estimation is a relatively simple method of determining the locations of spatial clusters based on point data.
- The default options (that we used) uses an algorithm to determine the size of each band (i.e., the distribution used) but you can change this if you want specific proportions for the bands (quartiles, etc.).
- Many ways of computing kernel densities in R

Comments on kernel density estimation

- For some datasets you will need to create a new SpatialPointsDataFrame by merging a non-spatial dataset with the coordinates from a SpatialLayer
  - More on this later…

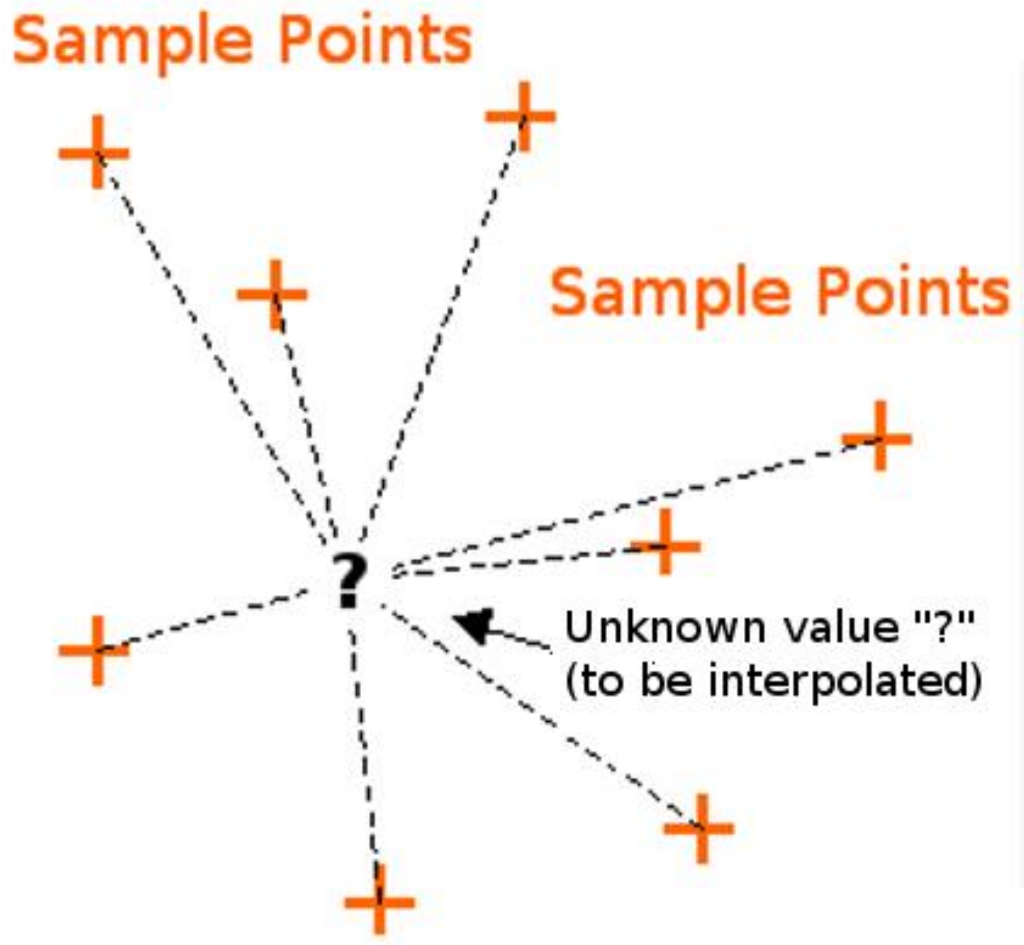
Inverse distance weighting

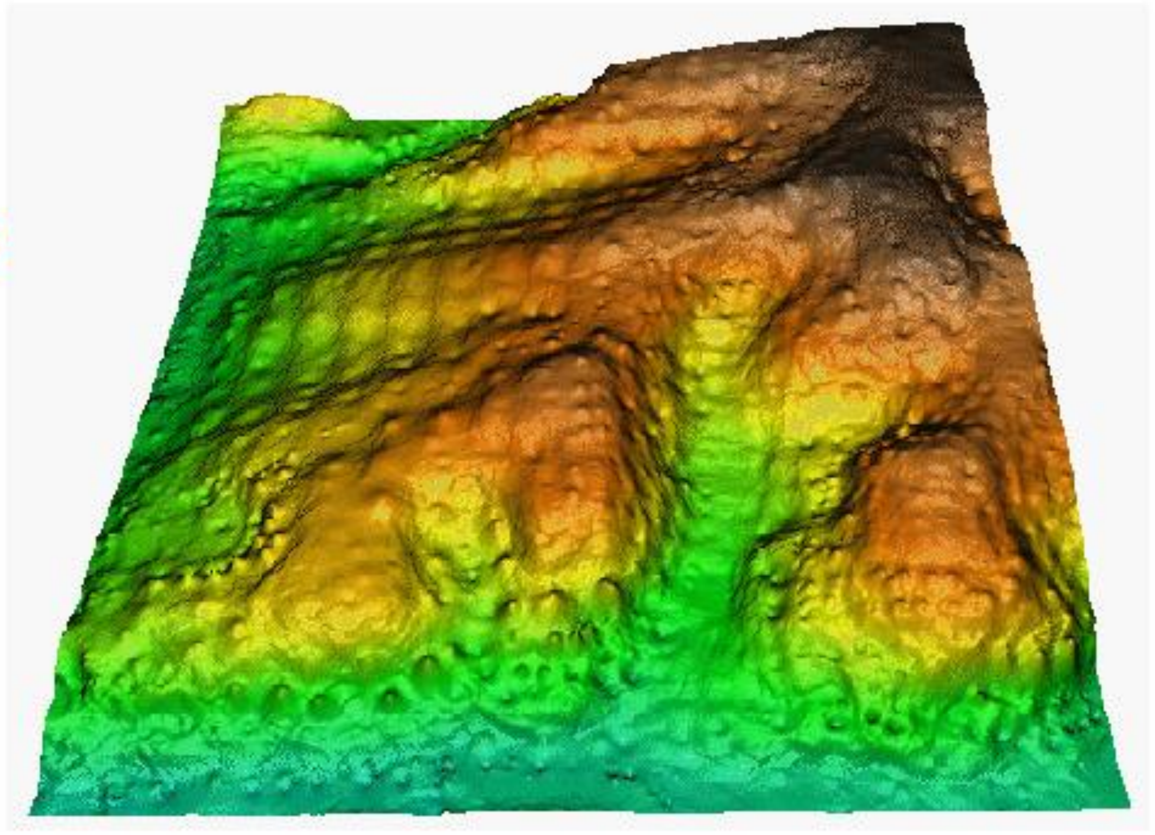
- Sometimes you may have point data with values for specific locations but want to (spatially) interpolate the value for areas between these locations.
- Frequently used for weather observations



Inverse distance weighting

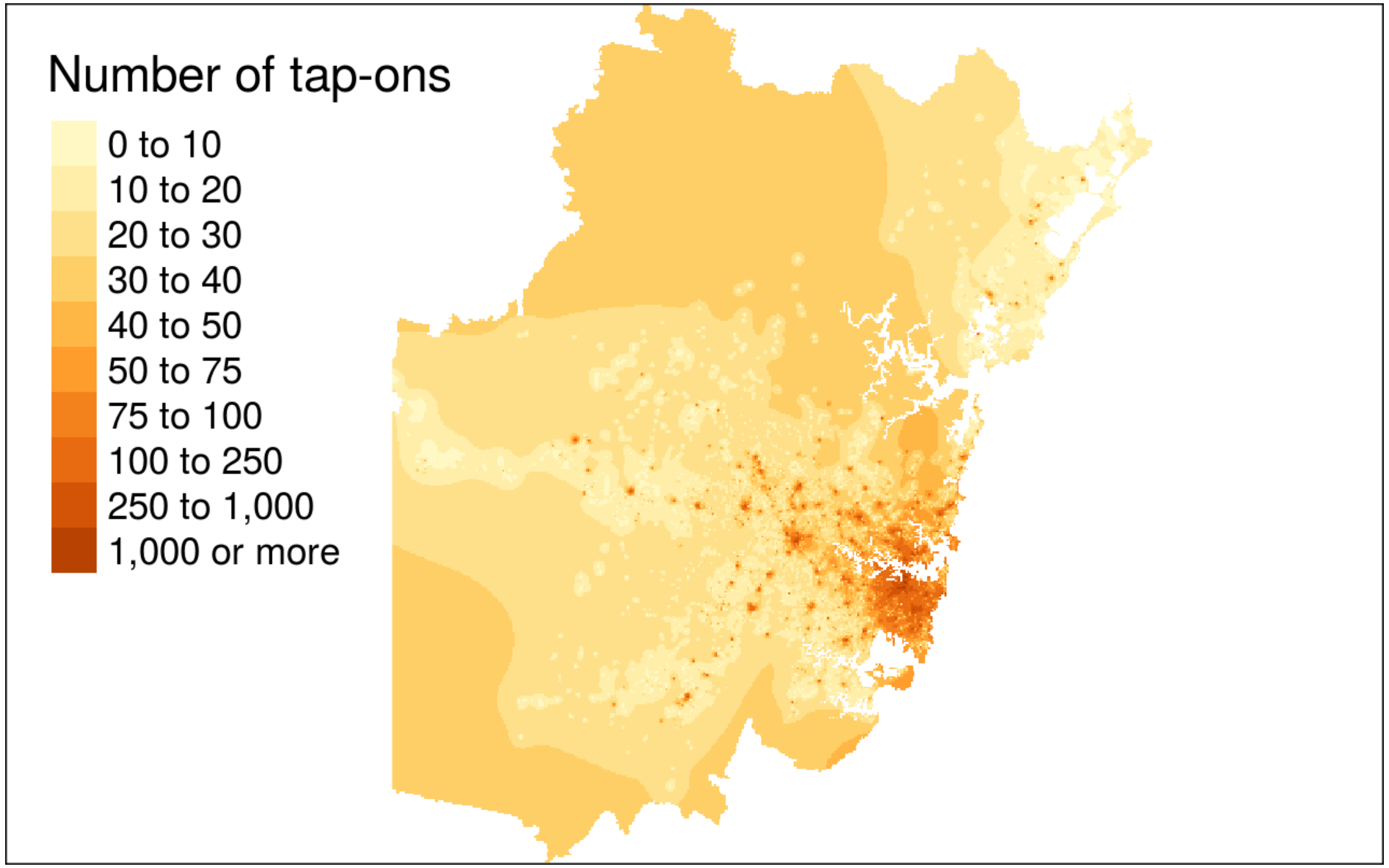
– Works by weighting the values of each known point by the distance to the unknown points the value is being calculated for.





Inverse distance weighting

– The same method can be used to create a plot that show continuous changes in values for point data.



Inverse distance weighting

- In R we can use the **idw** function in the **gstat** package to run an inverse distance weighting procedure
- Recall that we loaded this package earlier
- We will need our opal data layer we used for the first assignment and need to project it to a projected coordinate system:

```
opaldata <- readOGR("E:/data/opaldata","sydney_tapon_nums")
opaldata <- spTransform(opaldata, get_proj4(3308)) # Transform to NSW Lambert Projection
```

Inverse distance weighting

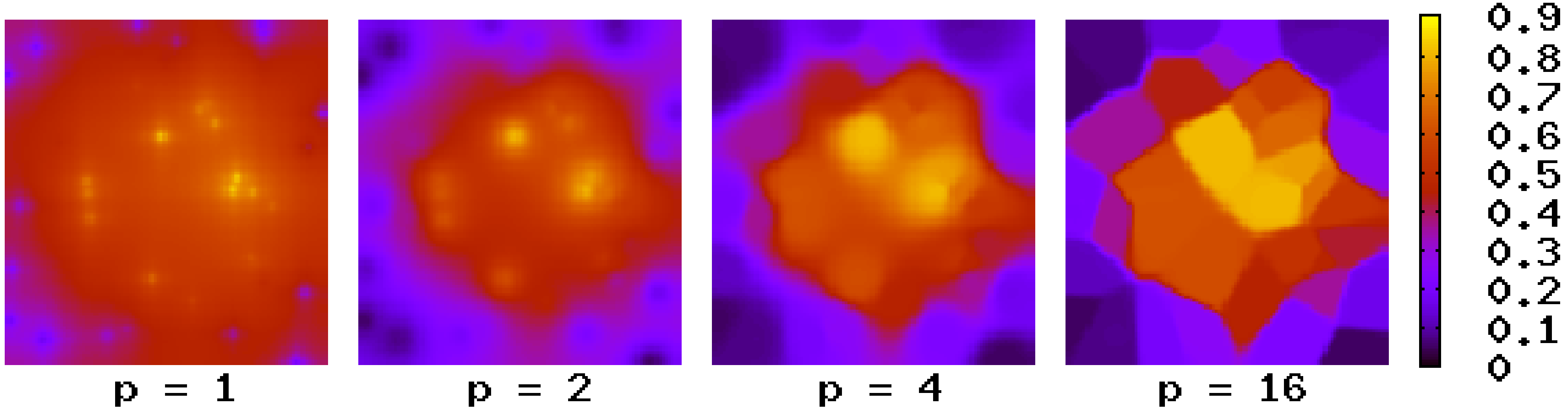
- To use the **idw** function we will need to create a new raster grid for points to interpolate
- In R we can assign a value to a function output, sometimes makes it easier to read and avoids copying the whole dataset to a new object

```r
# Create a new table (i.e., "Data Frame") with the coordinates of our new raster layer
rastergrid <- as.data.frame(spsample(opaldata, "regular", n=250000))
# Change the names of our columns to "X" and "Y"
names(rastergrid) <- c("X","Y")
# Tell R that our data.frame should be a SpatialLayer
coordinates(rastergrid) <- c("X", "Y")
# Tell R that our points are a point grid
gridded(rastergrid) <- TRUE
# Tell R that we want a full grid (not just points)
fullgrid(rastergrid) <- TRUE
# Set the projection to the projection of our opal layer
proj4string(rastergrid) <- proj4string(opaldata)
```

Inverse distance weighting

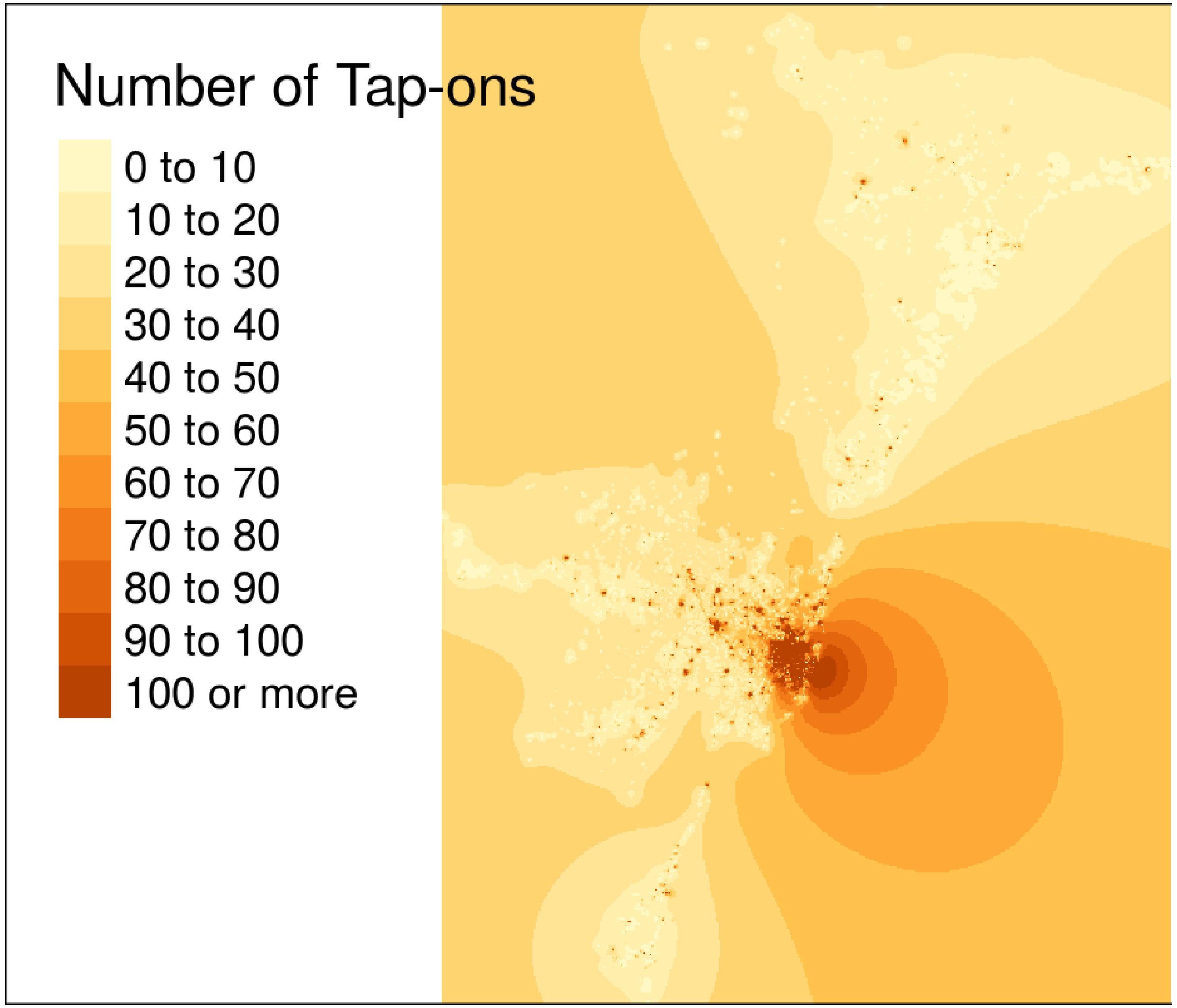
- Running the IDW function requires us to tell R how to calculate the unknown values
- We do this by specifying a function where the ~ means "is a function of"
- We can then convert the result into a raster layer

```r
opalidw <- idw(num_tapons ~ 1, opaldata, newdata=rastergrid, idp = 2.0)
opalidwr <- raster(opalidw)
```



Inverse distance weighting

- Plotting the results shows that what is returned is a raster layer that covers the whole grid
- Completely ignores geography!



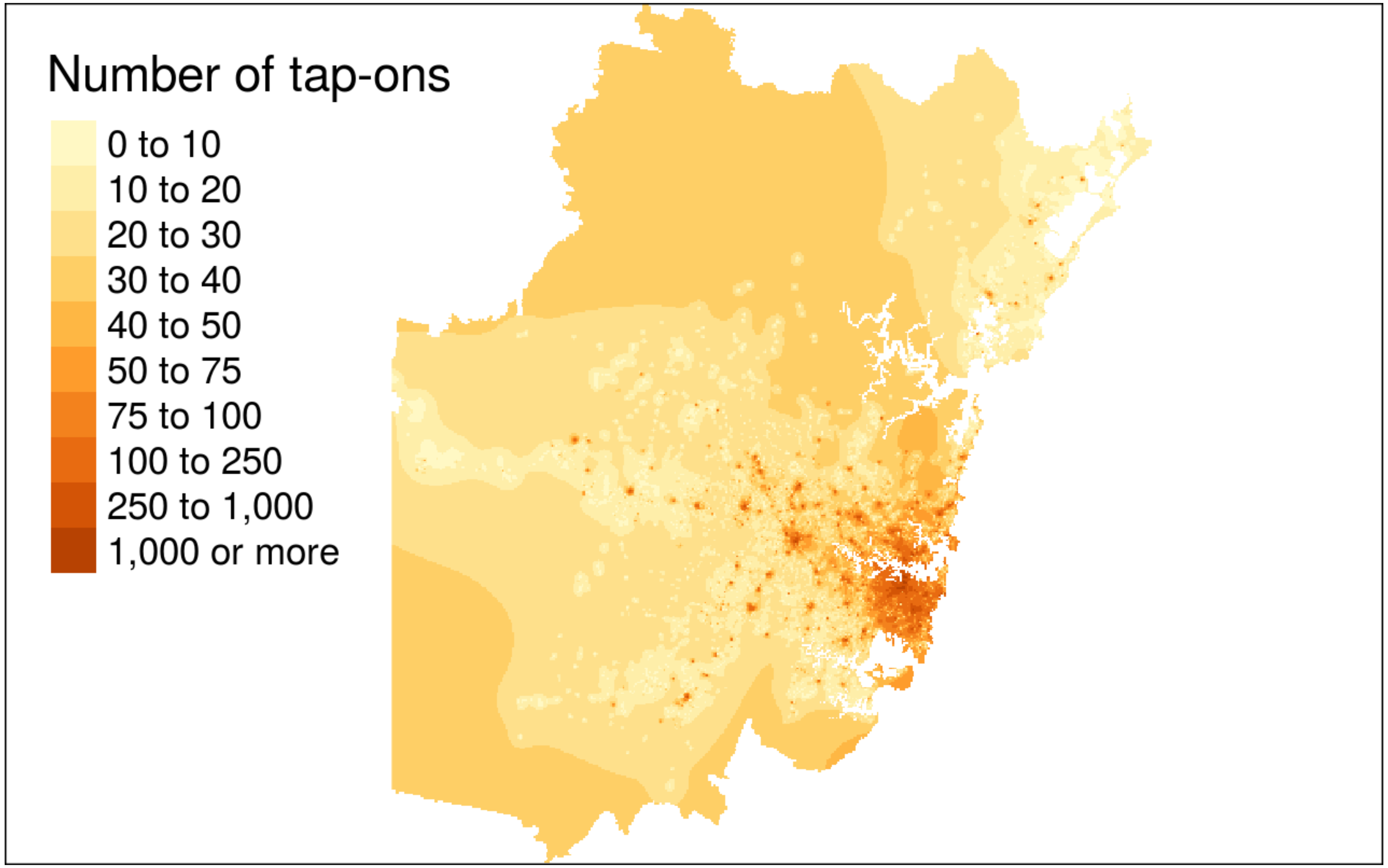
Inverse distance weighting

- We need to remove areas outside our geographic boundary using the **mask** function from the **raster** package
  - Plot the result and set different colours based on the ranges in the “breaks” argument:

```
opalidwrcrop <- mask(opalidwr, sydsa4)
tm_shape(opalidwrcrop) +
       tm_raster(breaks = c(0,10,20,30,40,50,75,100,250,1000,Inf))
```

Inverse distance weighting

- Now respects (somewhat) the geography



Inverse distance weighting

– If necessary we can convert our raster layer to a line layer:

```
opalcontours <- rasterToContour(
        opalidwr,
        levels=c(0,10,20,30,40,50,75,100,250,1000,Inf)
)

tm_shape(opalcontours) +
        tm_lines(col="level")
```

Final questions on kernel densities or IDW?

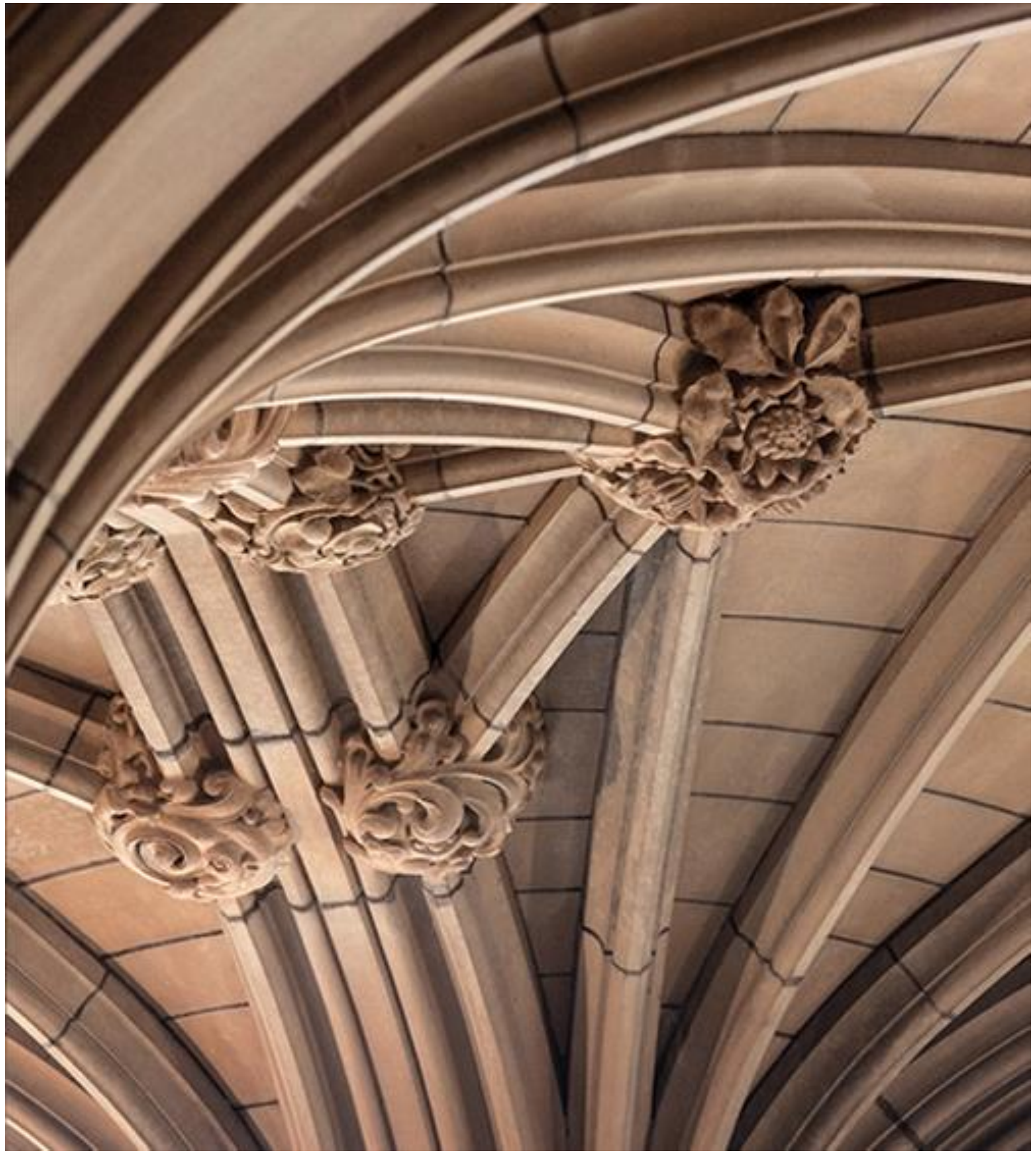
Databases with R and ArcGIS

Dr Adrian Ellison

Dr Richard Ellison

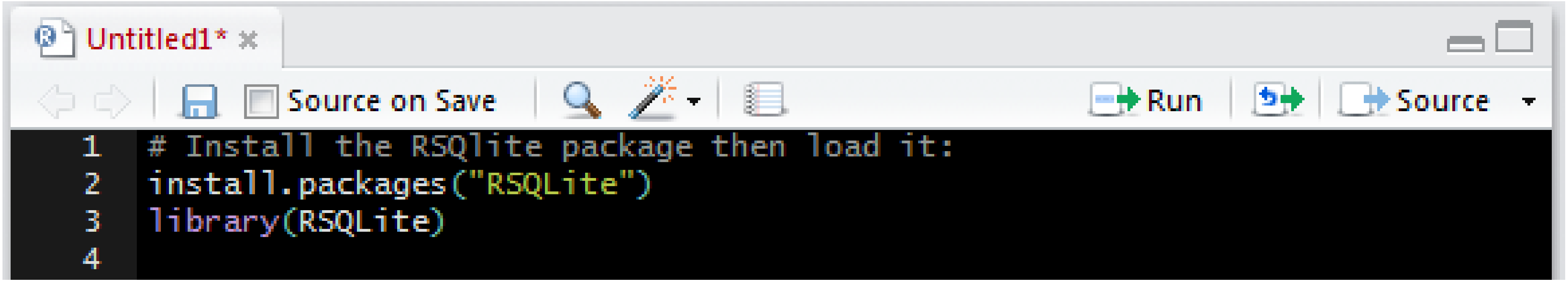
ITLS6107 Semester 2, 2017

THE UNIVERSITY OF SYDNEY



Install the RSQLite package

- For this session we will be using the RSQLite package
- Recall that to install and load a package in R we run:



```r
# Install the RSQlite package then load it:
install.packages("RSQLite")
library(RSQLite)

```

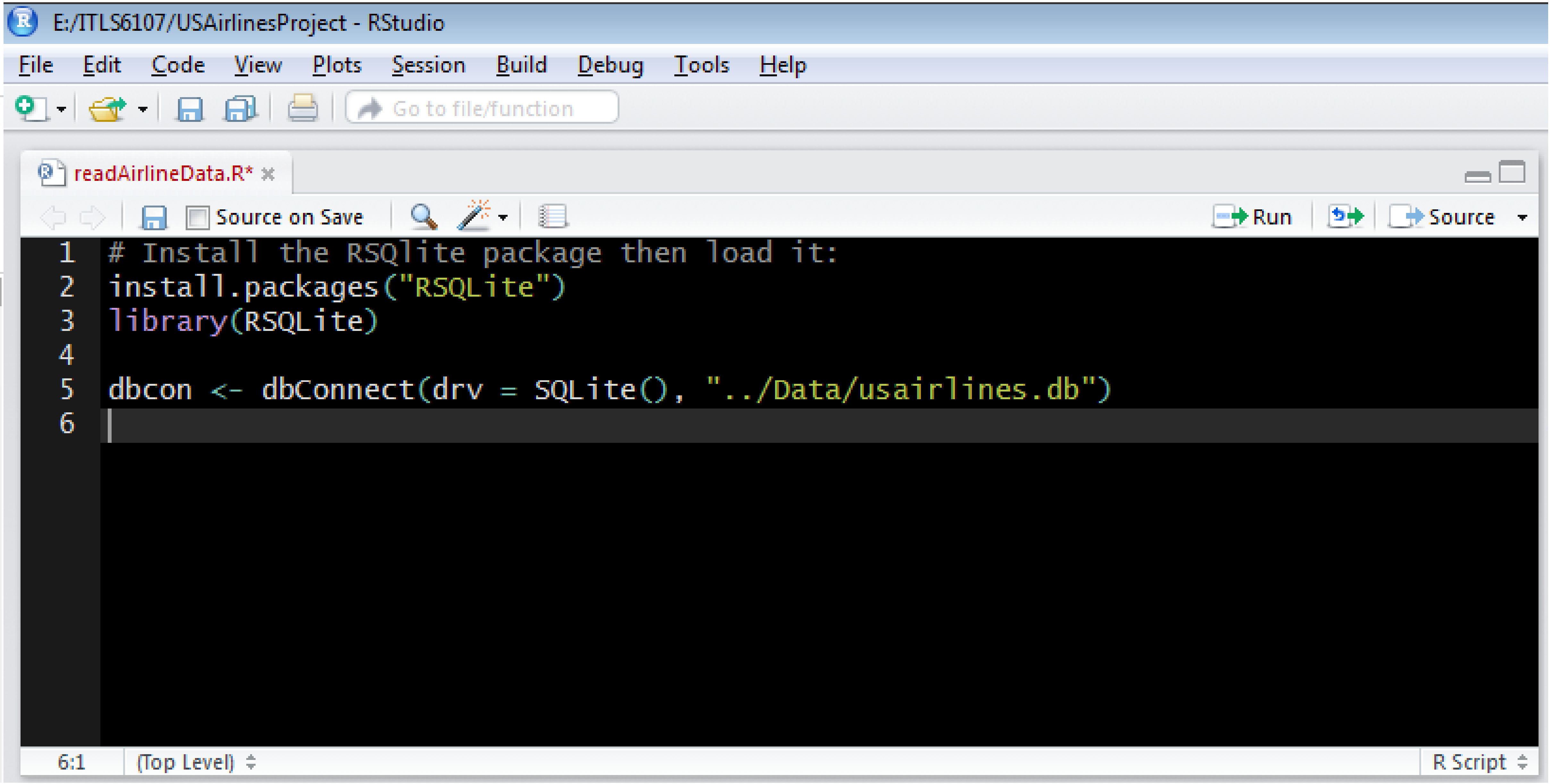
SQLite and RSQLite

- SQLite is a flat-file database that supports using SQL queries as if it were a standard relational database
  - Frequently used for small databases where data integrity is not important (e.g., iPhone and Android apps)
  - Not a good idea to use it for very large or complex databases
  - Has some restrictions in how it can be used
    - Not able to delete fields!
- Main advantages are:
  - Quick to setup
  - No need to install anything special to use it
  - Can run spatial queries using the Spatialite extension (needs installation)

SQLite and RSQLite

- The RSQLite package is used to access an SQLite database in R
  - Similar packages are available for standard relational databases:
    - RMySQL (for MySQL)
    - RPostgreSQL (for PostgreSQL)
    - Etc.
  - All are used in the same way (with a few minor differences)

Connecting to the database


E:/ITLS6107/USAirlinesProject - RStudio
File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
readAirlineData.R*
Source on Save
Run
Source
6:1
(Top Level)
R Script

```r
# Install the RSQlite package then load it:
install.packages("RSQLite")
library(RSQLite)

dbcon <- dbConnect(drv = SQLite(), "../Data/usairlines.db")
```

US domestic flights database

- For this session we will be using a database that contains all domestic flights in the United States in November 2015.
  - Contains 467,972
- The full dataset contains all flights from 1987 to (currently) November 2015
  - Approximately 130 million flights
- Two tables:
  - flights – Table containing all the flight data
  - airports – Table containing all the airports (the same as the .csv file on Blackboard)

flights table

- Year/Month/DayOfMonth/DayOfWeek – Columns for the date of the flight (all integers)
- ArrTime/DepTime – Actual arrival and departure times in 24 hour times as 4 digit numbers
- CRSArrTime/CRSDepTime – Scheduled arrival and departure times
- UniqueCarrier – IATA code for the airline
- FlightNum – Flight number (excluding the airline code)
- TailNum – Identifier of the aeroplane
- ArrDelay – Minutes the flight was delayed (negative if early)
- Origin/Dest –IATA Airport code of origin/destination
- Others

Interacting with the database

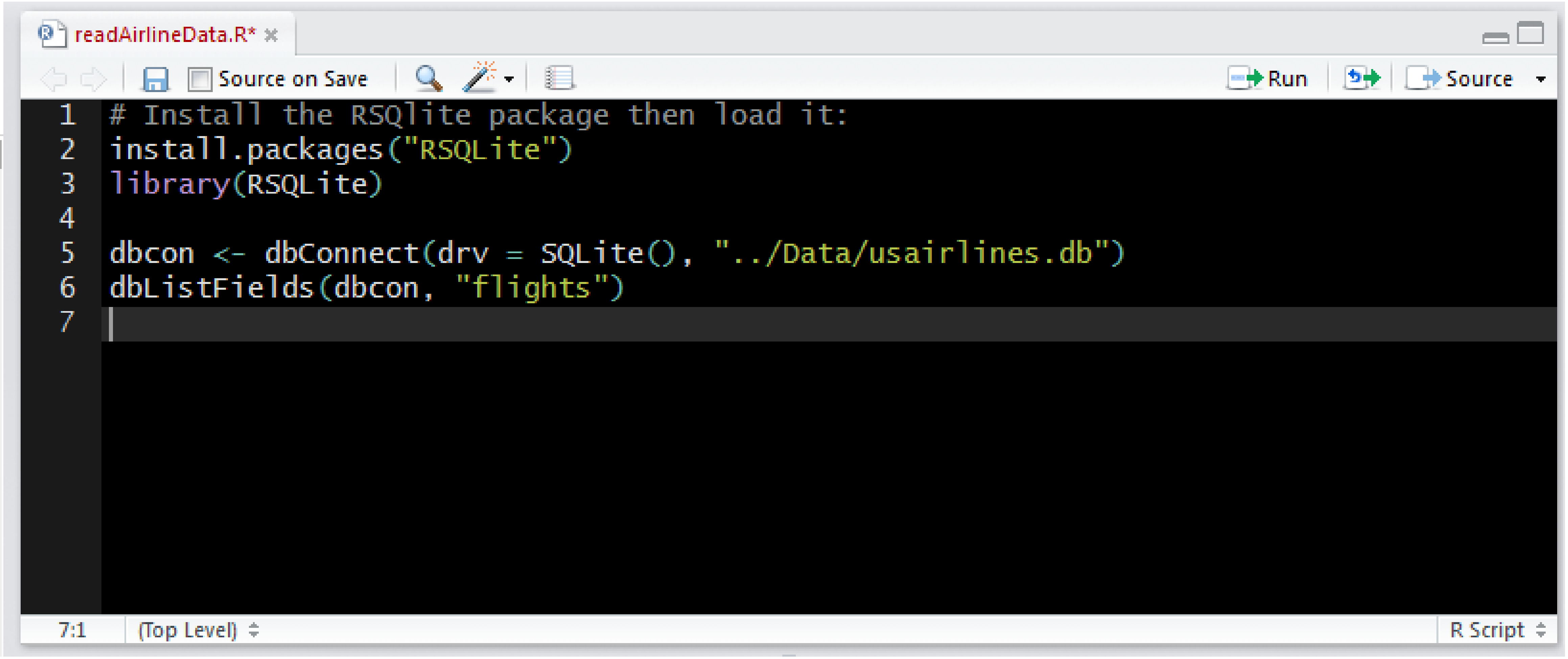
- RSQLite includes several methods for interacting with SQLite
  - dbListTables() – List the tables in the database
  - dbFetch() – Used to get results in sets (rather than all at once)
  - dbGetQuery() – Used to get all the results of a query at the same time
  - dbReadTable() – Read a whole table into R
  - dbGetRowCount() – Tells you many rows there are in your result
  - Others

Interacting with the database

- We will mostly use dbGetQuery() but dbFetch() is useful if you want to download large amounts of data
- dbGetQuery() returns the results of your query as a standard R dataframe
- Need to think carefully about what you actually need to retrieve from the database
  - Do you need the individual records or do you want to run aggregate analyses?

Showing a full list of fields in the database

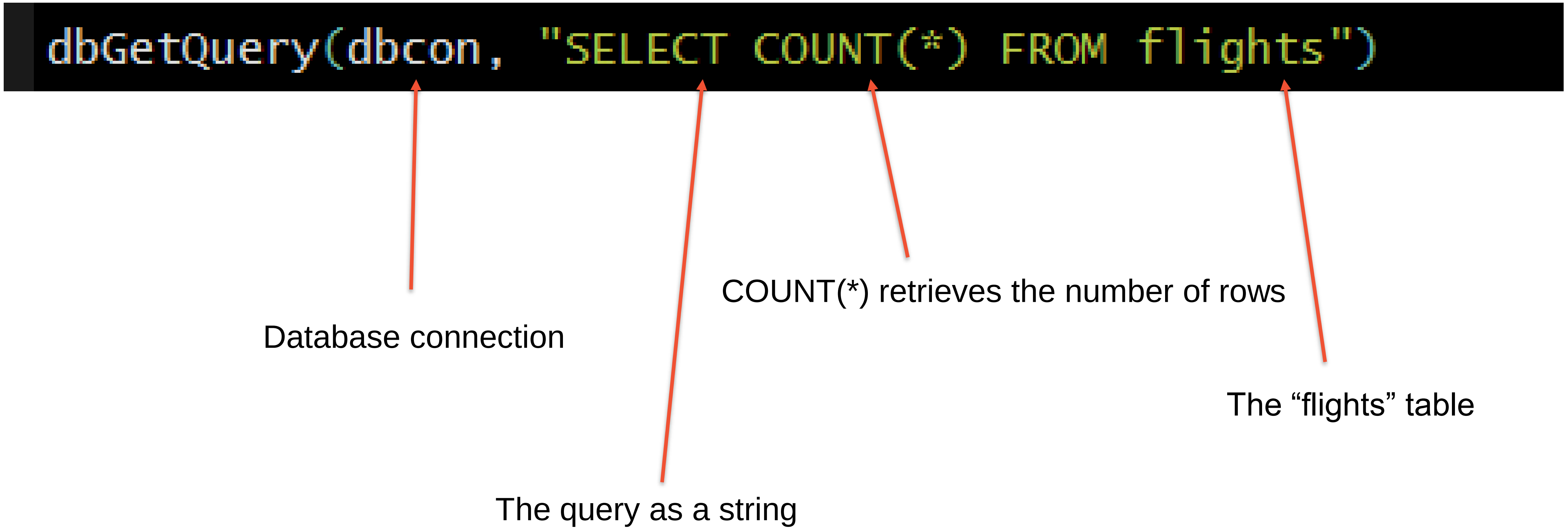
– Once we have connected to the database we can get more information about the database:



```r
# Install the RSQlite package then load it:
install.packages("RSQLite")
library(RSQLite)

dbcon <- dbConnect(drv = SQLite(), "../Data/usairlines.db")
dbListFields(dbcon, "flights")
```

Running a simple query on the database

– The dbGetQuery() function is used to query the database:

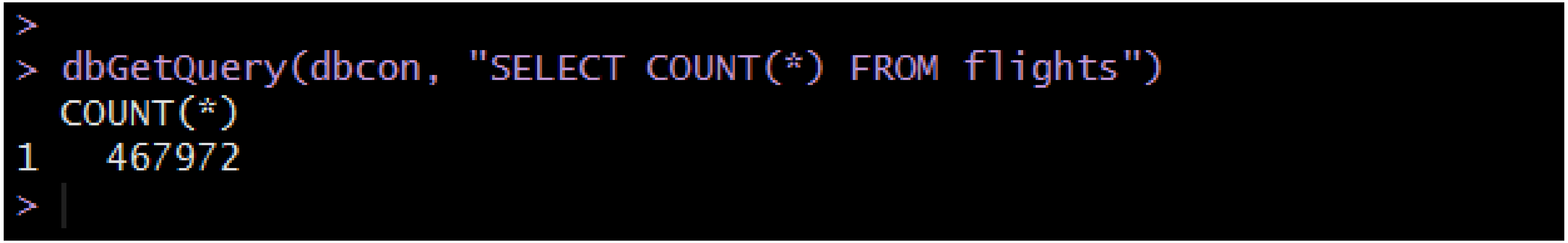


```
dbGetQuery(dbcon, "SELECT COUNT(*) FROM flights")
```

– Remember that the standard requires double speech marks around field names – SQLite does not require this

Running a simple query on the database

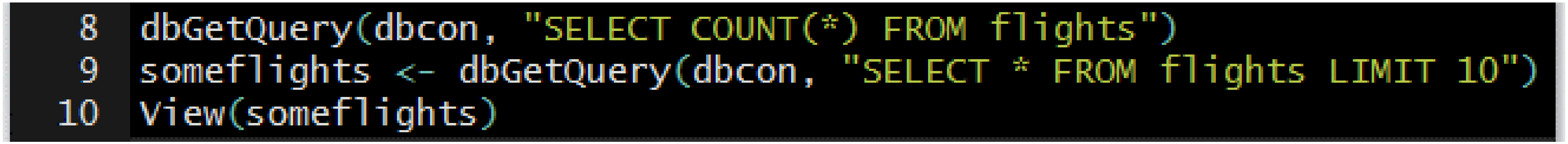
– Tells us the number of rows in our flights table:



```
>
> dbGetQuery(dbcon, "SELECT COUNT(*) FROM flights")
  COUNT(*)
1   467972
>
```

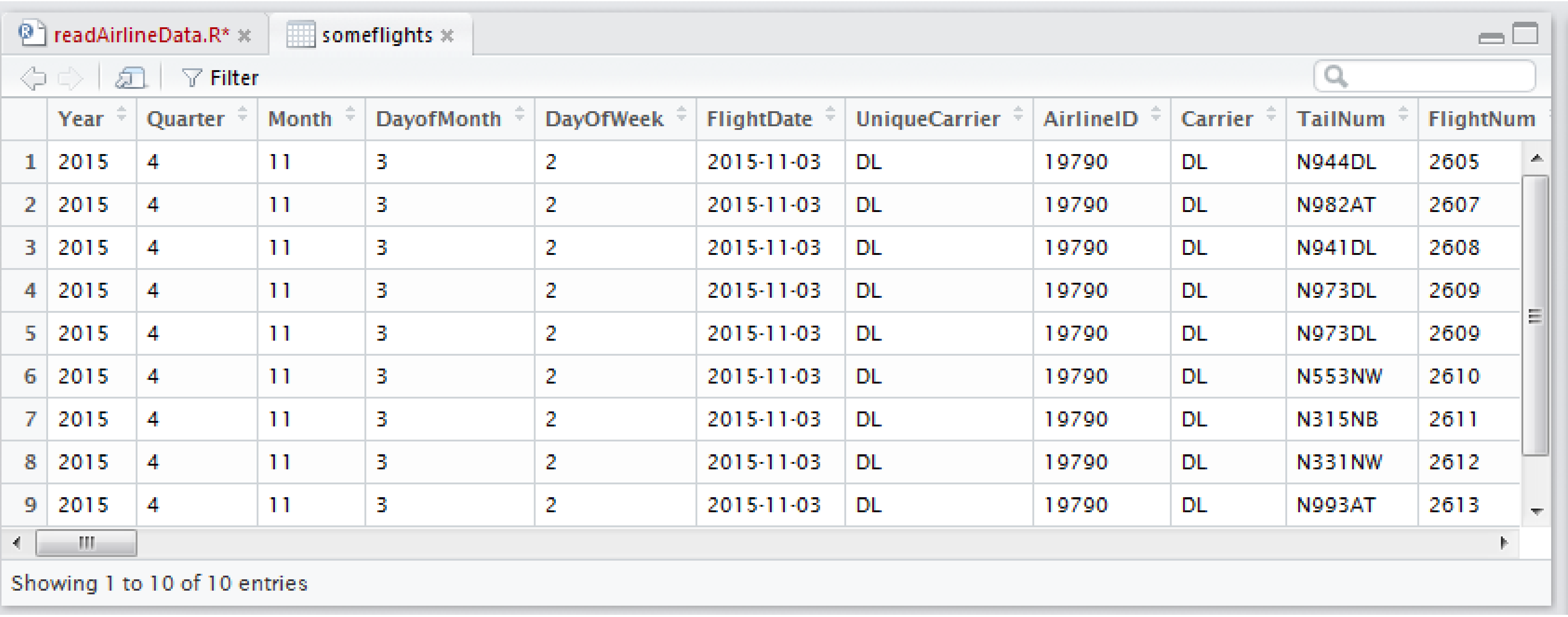
Running a simple query on the database

- Remember, to retrieve all the fields in a table we use a SELECT * FROM… query
  - LIMIT says to only return the first x observations



```
dbGetQuery(dbcon, "SELECT COUNT(*) FROM flights")
someflights <- dbGetQuery(dbcon, "SELECT * FROM flights LIMIT 10")
View(someflights)
```

Running a simple query on the database



	Year	Quarter	Month	DayofMonth	DayOfWeek	FlightDate	UniqueCarrier	AirlineID	Carrier	TailNum	FlightNum
1	2015	4	11	3	2	2015-11-03	DL	19790	DL	N944DL	2605
2	2015	4	11	3	2	2015-11-03	DL	19790	DL	N982AT	2607
3	2015	4	11	3	2	2015-11-03	DL	19790	DL	N941DL	2608
4	2015	4	11	3	2	2015-11-03	DL	19790	DL	N973DL	2609
5	2015	4	11	3	2	2015-11-03	DL	19790	DL	N973DL	2609
6	2015	4	11	3	2	2015-11-03	DL	19790	DL	N553NW	2610
7	2015	4	11	3	2	2015-11-03	DL	19790	DL	N315NB	2611
8	2015	4	11	3	2	2015-11-03	DL	19790	DL	N331NW	2612
9	2015	4	11	3	2	2015-11-03	DL	19790	DL	N993AT	2613

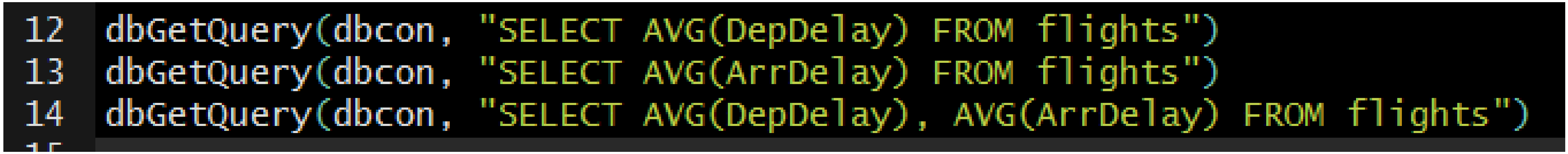
Showing 1 to 10 of 10 entries

Calculating summary statistics using the database

- Using a database provides a powerful method of quickly calculating summary statistics without needing to import the data into R:
- SQLite provides functions for some summary statistics
  - Average: AVG()
  - Minimum: MIN()
  - Maximum: MAX()
  - Sum: SUM()
  - Number of observations: COUNT()
- Other databases provide many more of the standard statistical functions
  - Median, Standard deviation, etc.

Calculating summary statistics using the database

- We can use different variations of the same query to look at the average values of different fields, or multiple fields in the same query:

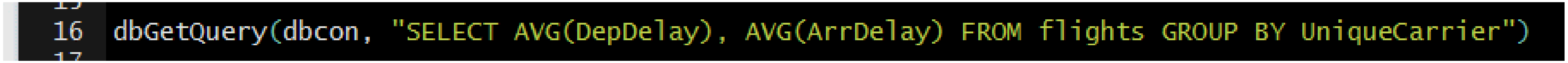


```
dbGetQuery(dbcon, "SELECT AVG(DepDelay) FROM flights")
dbGetQuery(dbcon, "SELECT AVG(ArrDelay) FROM flights")
dbGetQuery(dbcon, "SELECT AVG(DepDelay), AVG(ArrDelay) FROM flights")
```

- These queries return a statistic for the whole table.
  - What if we want to look at the average delay by airline?

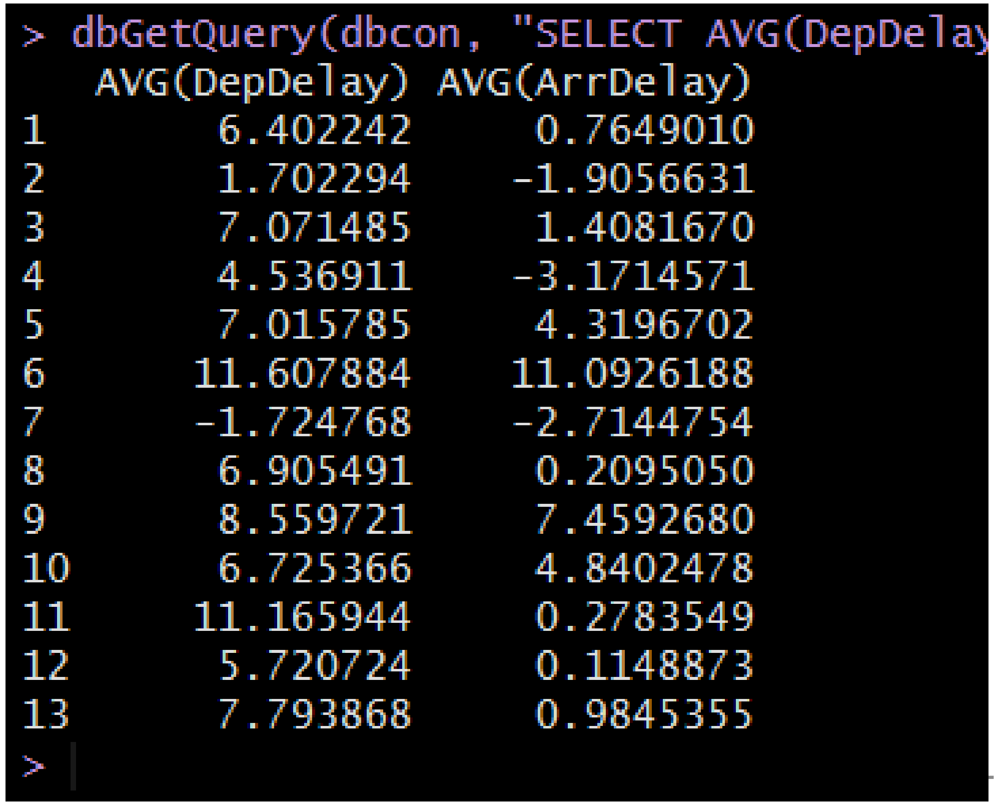
Calculating summary statistics using the database

- We use the GROUP BY clause to calculate different statistics for each value of the field(s) listed after the GROUP BY clause.



```
dbGetQuery(dbcon, "SELECT AVG(DepDelay), AVG(ArrDelay) FROM flights GROUP BY UniqueCarrier")
```

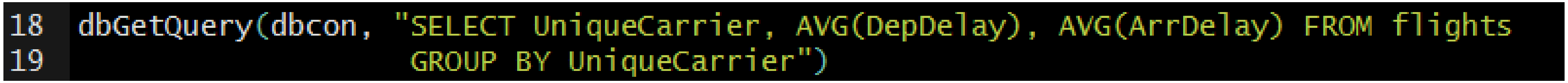
- Gives us exactly the same output but separated by airline.
- Problem is, which airline is which?



```
> dbGetQuery(dbcon, "SELECT AVG(DepDelay
   AVG(DepDelay) AVG(ArrDelay)
1       6.402242     0.7649010
2       1.702294    -1.9056631
3       7.071485     1.4081670
4       4.536911    -3.1714571
5       7.015785     4.3196702
6      11.607884    11.0926188
7      -1.724768    -2.7144754
8       6.905491     0.2095050
9       8.559721     7.4592680
10      6.725366     4.8402478
11     11.165944     0.2783549
12      5.720724     0.1148873
13      7.793868     0.9845355
> 
```

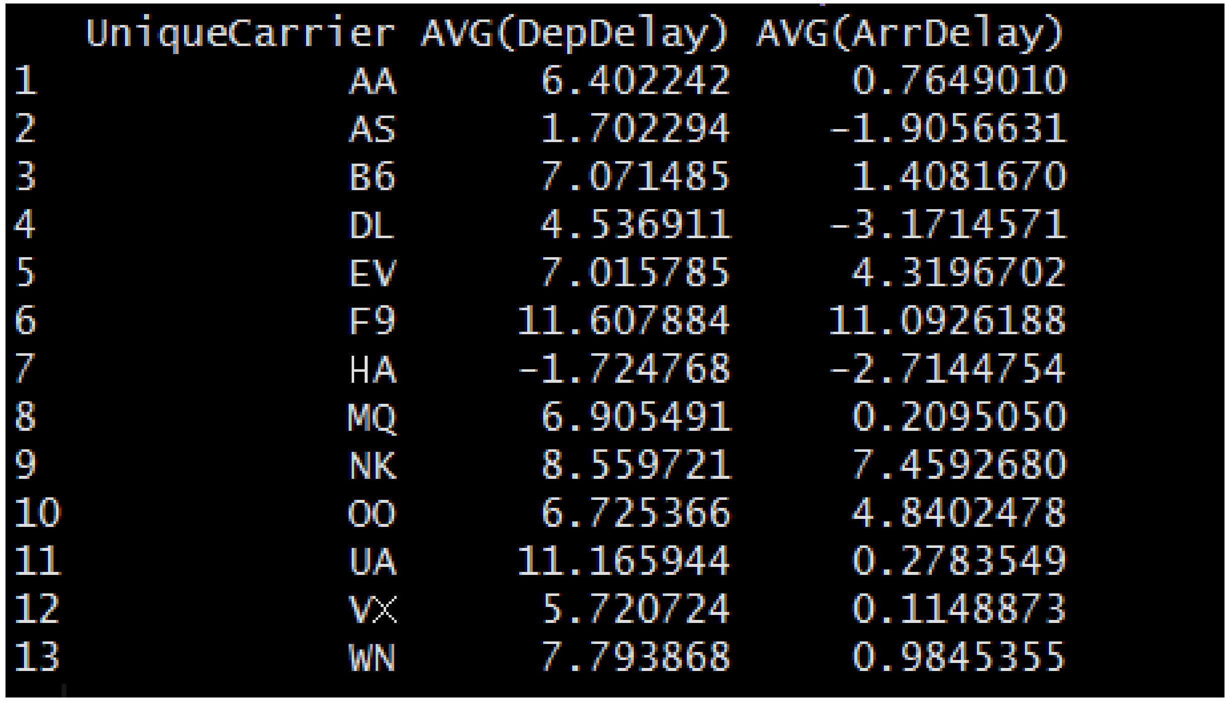
Calculating summary statistics using the database

- We use the GROUP BY clause to calculate different statistics for each value of the field(s) listed after the GROUP BY clause.



```
dbGetQuery(dbcon, "SELECT UniqueCarrier, AVG(DepDelay), AVG(ArrDelay) FROM flights
                    GROUP BY UniqueCarrier")
```

- We tell the database to return UniqueCarrier as a column in our results
- We can change the UniqueCarrier field to another field name to look at
  - Airports
  - Day of the month
  - Day of the week
  - Aircraft



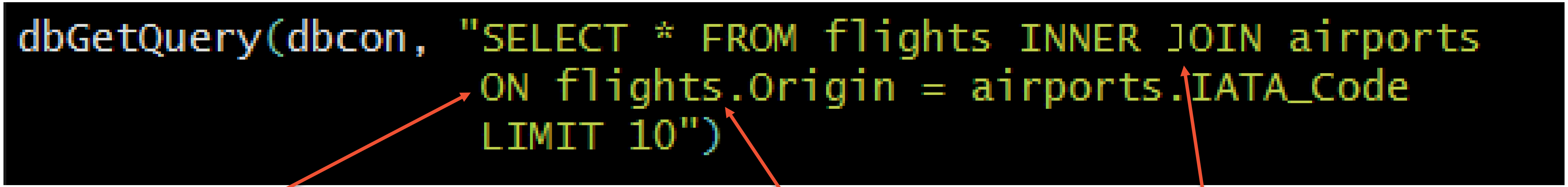
```
   UniqueCarrier AVG(DepDelay) AVG(ArrDelay)
1             AA      6.402242     0.7649010
2             AS      1.702294    -1.9056631
3             B6      7.071485     1.4081670
4             DL      4.536911    -3.1714571
5             EV      7.015785     4.3196702
6             F9     11.607884    11.0926188
7             HA     -1.724768    -2.7144754
8             MQ      6.905491     0.2095050
9             NK      8.559721     7.4592680
10            OO      6.725366     4.8402478
11            UA     11.165944     0.2783549
12            VX      5.720724     0.1148873
13            WN      7.793868     0.9845355
```

Using joins in the database

- In ArcGIS, using a (non-spatial) join allows us to link two tables by a common attribute.
  - ArcGIS does this by querying the File Geodatabase tables we want to join.
- The equivalent procedure can be run using SQL

Using joins in the database

- In ArcGIS, using a (non-spatial) join allows us to link two tables by a common attribute.
  - ArcGIS does this by querying the File Geodatabase tables we want to join.
- The equivalent procedure can be run using SQL



```
dbGetQuery(dbcon, "SELECT * FROM flights INNER JOIN airports
                   ON flights.Origin = airports.IATA_Code
                   LIMIT 10")
```

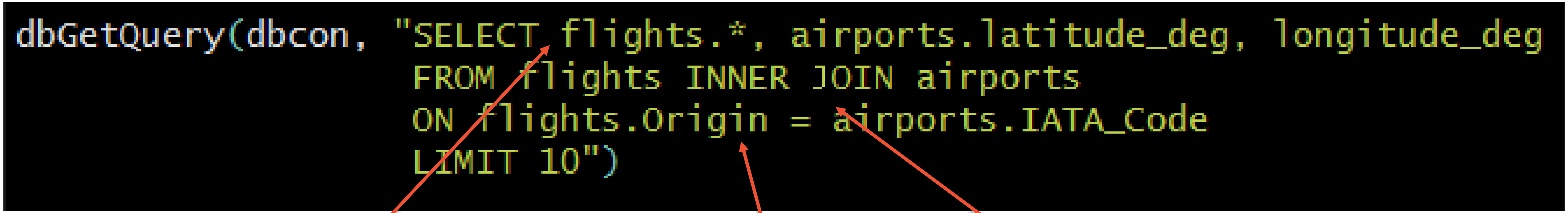
ON tells us what attributes to join by

Attributes in the ON clause are specified by the format: TableName.FieldName

INNER JOIN tells the database to only return matching rows

Using joins in the database

– If we want all fields from the flights table but only some from the airports table we can change our SELECT statement to:



```
dbGetQuery(dbcon, "SELECT flights.*, airports.latitude_deg, longitude_deg
                   FROM flights INNER JOIN airports
                   ON flights.Origin = airports.IATA_Code
                   LIMIT 10")
```

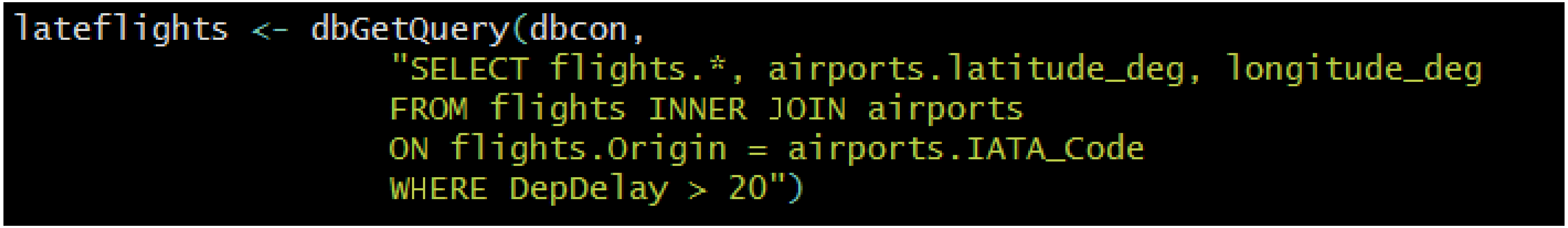
Fields in the SELECT statement are specified the same way as in the ON clause.
We can also specify them without the table name if the field name is unique in both tables.

Attributes in the ON clause are specified by the format: TableName.FieldName

INNER JOIN tells the database to only return matching rows

Are there clusters of delayed flights?

- If we only want to look at flights that depart more than 20 minutes (or another value) late, we can use the WHERE clause to retrieve only flights that match that condition:

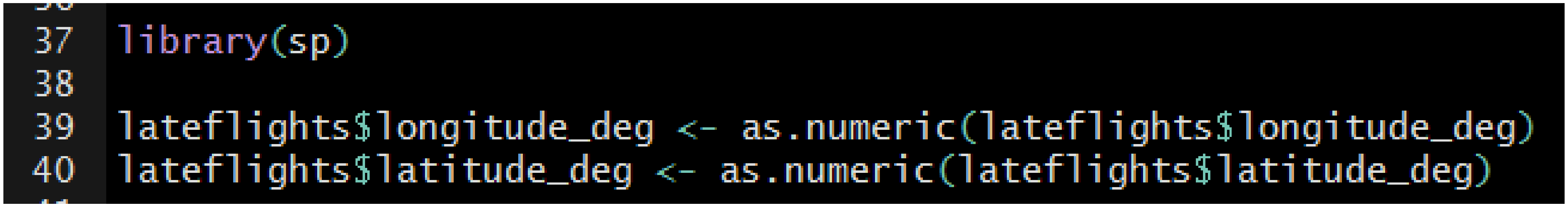


```
lateflights <- dbGetQuery(dbcon,
                 "SELECT flights.*, airports.latitude_deg, longitude_deg
                 FROM flights INNER JOIN airports
                 ON flights.Origin = airports.IATA_Code
                 WHERE DepDelay > 20")
```

- We are saving the results of our query to the (new) lateflights dataframe

Are there clusters of delayed flights?

- Once we have our data in R we need to convert it to a spatial format R understands
- We are looking at the origin airports only so we are using point data
- Remember to load the sp and sf packages
- Our data was imported from SQLite as character data but we need the coordinates as numeric so we also need to convert them:



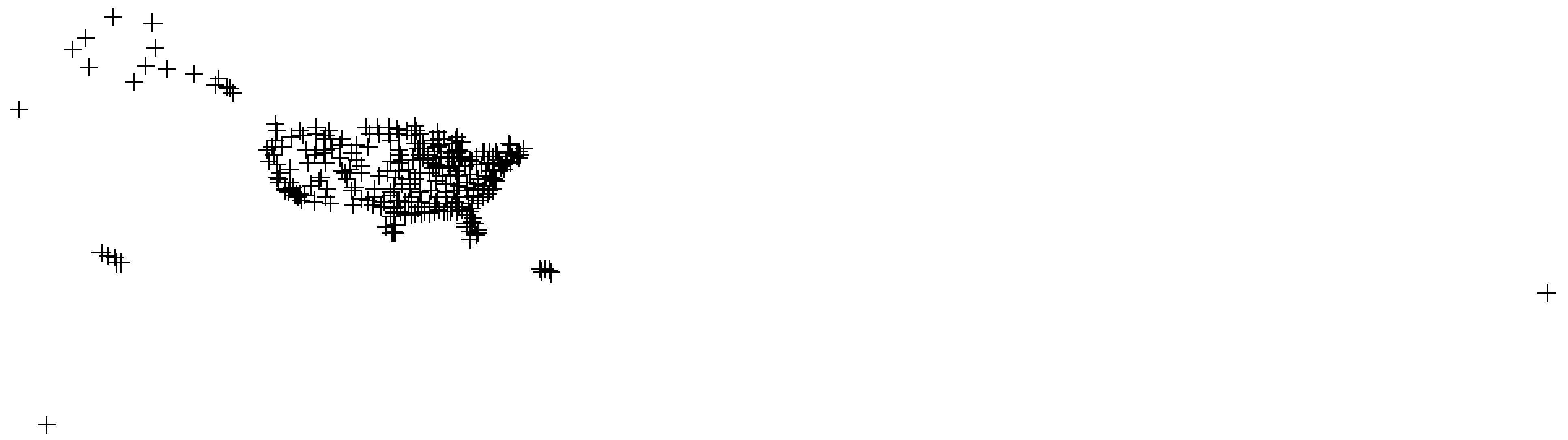
```
library(sp)

lateflights$longitude_deg <- as.numeric(lateflights$longitude_deg)
lateflights$latitude_deg <- as.numeric(lateflights$latitude_deg)
```

Are there clusters of delayed flights?

- We then create our spatial object using the st_as_sf() function.
- Try plotting the points

Are there clusters of delayed flights?

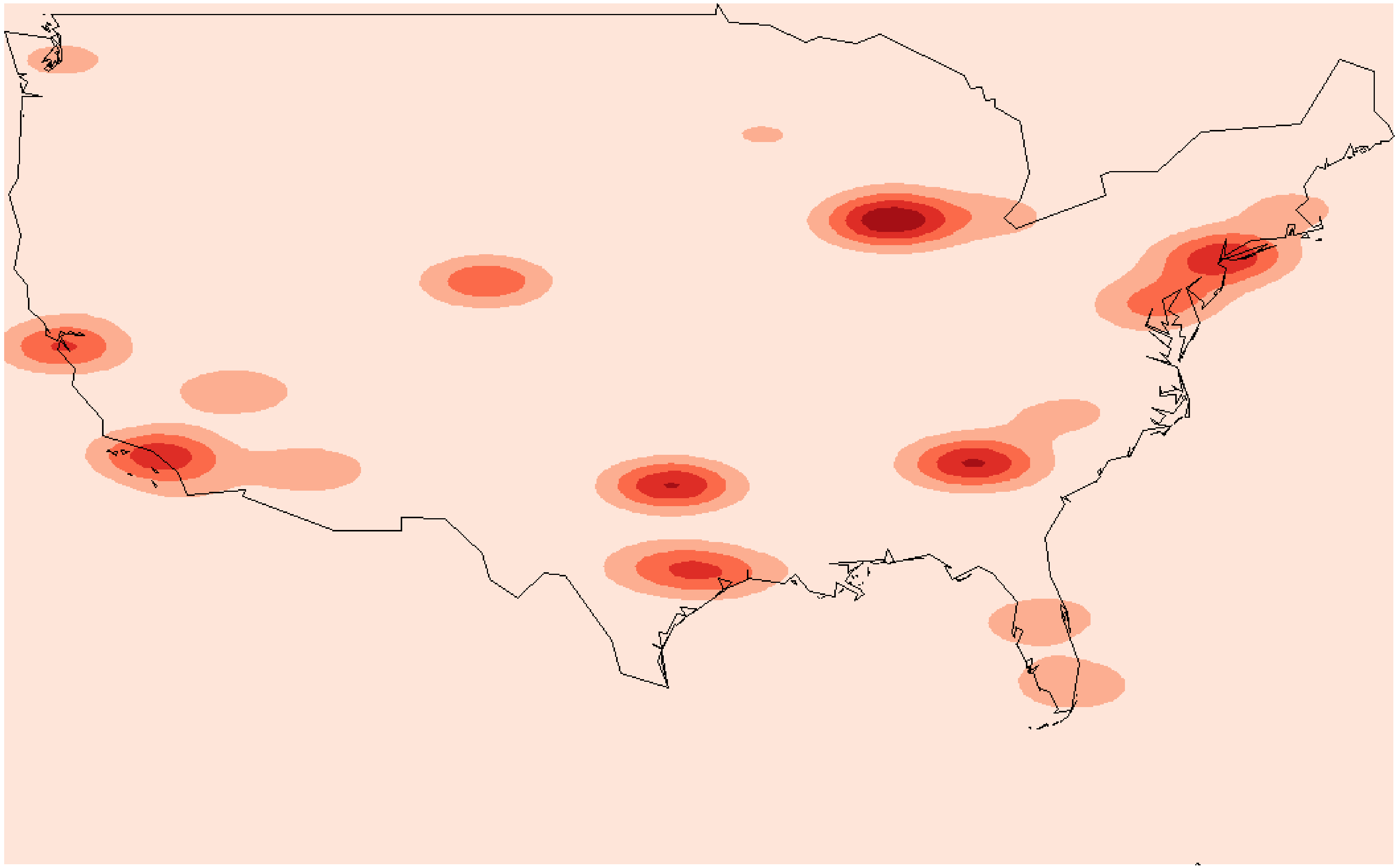
- United States is recognisable but there are some outliers
  - Which airports are these?



Are there clusters of delayed flights?

- We want to know where these delayed flights occurred
  - Are they concentrated in a specific area or more spread out?

Are there clusters of delayed flights?



Other methods of interacting with databases in R

- The dplyr package provides functionality to interact with databases using its standard functions:
  - group_by()
  - mutate()
  - summarise()
  - Others…
- The postGIStools package provides functions for working with spatial Postgres databases that automatically converts the result into a spatial table in R and the st_read function can also read from spatial databases.