STATS SOLUTIONS

COUNT YOUR INTEREST BY STATISTICS.

The Tidyverse Cookbook

STATS_SOLUTIONS@ryannthegeek

2023-06-12

Contents

List of Figures

List of Tables

Tip

Packages used

`tidyverse, repurrsive`

1 Program

- Combine functions into a pipe

```
require(tidyverse)
starwars |>
  group_by(species) |>
  summarise(avg_height = mean(height, na.rm=TRUE)) |>
  arrange(avg_height) |>
  head()
```

```
# A tibble: 6 x 2
  species        avg_height
  <chr>               <dbl>
1 Yoda's species         66
2 Aleena                 79
3 Ewok                   88
4 Vulptereen             94
5 Dug                   112
6 Xexto                 122
```

2 Import

When you import data into R, R stores the data in your computer's RAM while you manipulate it. This creates a size limitation: truly big data sets should be stored outside of R in a database or a distributed storage system. You can then create a connection to the system that R can use to access the data without bringing the data into your computer's RAM.

The readr package contains the most common functions in the tidyverse for importing data. The readr package is loaded when you run library(tidyverse). The tidyverse also includes the following packages for importing specific types of data. These are not loaded with library(tidyverse). You must load them individually when you need them.

1. DBI - connect to databases
2. haven - read SPSS, Stata, or SAS data
3. httr - access data over web APIs
4. jsonlite - read JSON
5. readxl - read Excel spreadsheets
6. rvest - scrape data from the web
7. xml2 - read XML

- Read a Compressed RDS file

```
saveRDS(pressure, file="file.RDS")

my_data <- readRDS("file.RDS")
head(my_data, 3)
```

```
  temperature pressure
1           0   0.0002
2          20   0.0012
3          40   0.0060
```

- Read an Excel spreadsheet

```
require(readxl)
file_path <- readxl_example('datasets.xlsx')
my_data <- read_excel(file_path)
head(my_data, 3)
```

```
# A tibble: 3 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl> <chr>
1          5.1         3.5          1.4         0.2 setosa
2          4.9         3            1.4         0.2 setosa
3          4.7         3.2          1.3         0.2 setosa
```

- Read a specific sheet from an Excel spreadsheet

```
my_data <- read_excel(file_path, sheet='chickwts')
head(my_data, 3)
```

```
# A tibble: 3 x 2
  weight feed
   <dbl> <chr>
1    179 horsebean
2    160 horsebean
3    136 horsebean
```

- Read a field of cells from an excel spreadsheet

```
my_data <- read_excel(file_path, range='C1:E4', skip=3, n_max=10)
head(my_data)
```

```
# A tibble: 3 x 3
  Petal.Length Petal.Width Species
         <dbl>       <dbl> <chr>
1          1.4         0.2 setosa
2          1.4         0.2 setosa
3          1.3         0.2 setosa
```

`read_excel()` adopts the skip and [n_max][Skip lines at the end of a file when reading a file] arguments of reader functions to skip rows at the top of the spreadsheet and to control how far down the spreadsheet to read. Use the `range` argument to specify a subset of columns to read. Two column names separated by a:‘ specifies those two columns and every column between them.

- Write to a comma-separate values (csv) file

```
write_csv(iris, file='my_file.csv')
```

- Write to a semi-colon delimited file; file that uses semi-colons to delimit cells

```
write_csv2(iris, file='my_file2.csv')
```

- Write to a tab-delimited file

```
write_tsv(iris, file='my_file3.tsv')
```

- Write to a text file with arbitrary delimiters

  You wanna save a tibble of df as a plain text file that uses an unusual delimiter

```
write_delim(iris, file='my_file4.tsv', delim='|')
```

- Write to a compressed RDS file

```
saveRDS(iris, file='my_file5.RDS')
```

3 Tidy

1. **Data tidying** refers to *reshaping* your data into a tidy data frame or tibble. Data tidying is an important first step for your analysis because every tidyverse function will expect your data to be stored as Tidy Data.

2. Tidy data is tabular data organized so that: Each column contains a single variable Each row contains a single observation.

3. A **variable** is a quantity, quality, or property that you can measure. An **observation** is a set of measurements made under similar conditions (you usually make all of the measurements in an observation at the same time and on the same object)

4. Tidy data is not an arbitrary requirement of the tidyverse; it is the ideal data format for doing data science with R.

5. Tidy data makes it easy to extract every value of a variable to build a plot or to compute a summary statistic.

- **Spread a pair of columns into a field of cells**; **pivot**, **convert long data to wide**, or move variable names out of the cells and into the column names.

  ```
  head(table2)
  ```

  ```
  # A tibble: 6 x 4
    country      year type            count
    <chr>       <dbl> <chr>           <dbl>
  1 Afghanistan  1999 cases             745
  2 Afghanistan  1999 population   19987071
  3 Afghanistan  2000 cases            2666
  4 Afghanistan  2000 population   20595360
  5 Brazil       1999 cases           37737
  6 Brazil       1999 population  172006362
  ```

  For example, `table2` contains `type`, which is a column that repeats the variable names `case` and `population`. To make `table2` tidy, you must move `case` and `population` values into their own columns.

```
table2 |>
  spread(key=type, value=count) |>
  head(3)
```

```
# A tibble: 3 x 4
  country      year cases population
  <chr>       <dbl> <dbl>      <dbl>
1 Afghanistan  1999   745   19987071
2 Afghanistan  2000  2666   20595360
3 Brazil       1999 37737  172006362
```

If you would to convert each new column to the most sensible data type given its final contents, add the argument `convert = TRUE`.

- **Gather a field of cells into a pair of columns**; **convert wide data to long**, reshape a **two-by-two table**, or move variable values out of the column names and into the cells.

  ```
  table4a
  ```

  ```
  # A tibble: 3 x 3
    country     `1999` `2000`
    <chr>        <dbl>  <dbl>
  1 Afghanistan    745   2666
  2 Brazil       37737  80488
  3 China       212258 213766
  ```

  For example, `table4a` is a two-by-two table with the column names `1999` and `2000`. These names are values of a `year` variable. The field of cells in `table4a` contains counts of TB cases, which is another variable. To make `table4a` tidy, you need to move year and case values into their own columns.

  ```
  table4a |>
    gather(key='year', value='cases', 2:3) # means gather values in
     column 2:3
  ```

  ```
  # A tibble: 6 x 3
    country     year   cases
    <chr>       <chr>  <dbl>
  1 Afghanistan 1999     745
  2 Brazil      1999   37737
  3 China       1999  212258
  4 Afghanistan 2000    2666
  5 Brazil      2000   80488
  6 China       2000  213766
  ```

  ```
  # you can add convert=T if you want to keep the changes permanently
  ```

- Separate a column into new columns

  ```
  table3
  ```

  ```
  # A tibble: 6 x 3
    country      year rate
    <chr>       <dbl> <chr>
  1 Afghanistan  1999 745/19987071
  2 Afghanistan  2000 2666/20595360
  ```

```
3 Brazil        1999 37737/172006362
4 Brazil        2000 80488/174504898
5 China         1999 212258/1272915272
6 China         2000 213766/1280428583
```

For example, `table3` combines `cases` and `population` values in a single column named `rate`. To tidy `table3`, you need to separate `rate` into two columns: one for the `cases` variable and one for the `population` variable

```
table3 |>
  separate(col=rate, into=c('cases', 'population'),
           sep='/', convert=TRUE)
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <dbl>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

- Unite multiple columns into a single column

```
table5
```

```
# A tibble: 6 x 4
  country     century year  rate
  <chr>       <chr>   <chr> <chr>
1 Afghanistan 19      99    745/19987071
2 Afghanistan 20      00    2666/20595360
3 Brazil      19      99    37737/172006362
4 Brazil      20      00    80488/174504898
5 China       19      99    212258/1272915272
6 China       20      00    213766/1280428583
```

```
table5 |>
  unite(col='year', century, year, sep='')
```

```
# A tibble: 6 x 3
  country     year  rate
  <chr>       <chr> <chr>
1 Afghanistan 1999  745/19987071
2 Afghanistan 2000  2666/20595360
3 Brazil      1999  37737/172006362
4 Brazil      2000  80488/174504898
5 China       1999  212258/1272915272
6 China       2000  213766/1280428583
```

4 Transform Tables

- The `dplyr` package provides the most important tidyverse functions for manipulating tables.

- `dplyr` functions always return a **transformed** *copy* of your table. They won't change your original table unless you tell them to (by saving over the name of the original table). That's good news, because you should always retain a clean copy of your original data in case something goes wrong.

- You can refer to columns by name inside of a dplyr function. There's no need for `$` syntax or `""`. Every dplyr function requires you to supply a data frame, and it will recognize the columns in that data frame, e.g.

  ```
  summarise(mpg, h = mean(hwy), c = mean(cty))
  ```

  This only becomes a problem if you'd like to use an object that has the same name as one of the columns in the data frame. In this case, place `!!` before the object's name to unquote it, `dplyr` will skip the columns when looking up the object. e.g.

  ```
  hwy <- 1:10
  summarise(mpg, h = mean(!!hwy), c = mean(cty))
  ```

- Transforming a table sometimes requires more than one recipe. Why? Because tables are made of multiple data structures that work together:

  1. The table itself is a data frame or tibble.
  2. The columns of the table are vectors.
  3. Some columns may be list-columns, which are lists that contain vectors.

- So to transform a table, begin with a recipe that transforms the structure of the table. You'll find those recipes in this chapter. Then complete it with a recipe that transforms the actual data values in your table. The Combine transform recipes recipe will show you how.

- Arrange rows by value in ascending order

  ```
  mpg |>
    arrange(displ) |>
    head()
  ```

  ```
  # A tibble: 6 x 11
    manufacturer model displ  year   cyl trans      drv     cty   hwy fl
         class
    <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <
     chr> <chr>
  1 honda        civic   1.6  1999     4 manual(m5) f        28    33 r
         subco~
  2 honda        civic   1.6  1999     4 auto(l4)   f        24    32 r
         subco~
  3 honda        civic   1.6  1999     4 manual(m5) f        25    32 r
         subco~
  4 honda        civic   1.6  1999     4 manual(m5) f        23    29 p
         subco~
  5 honda        civic   1.6  1999     4 auto(l4)   f        24    32 r
         subco~
  6 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p
         compa~
  ```

If you provide additional column names, `arrange()` will use the additional columns in order as tiebreakers to sort within rows that share the same value of the first column.

```
mpg |>
  arrange(displ, cty) |>
  head()
```

```
# A tibble: 6 x 11
  manufacturer model      displ  year   cyl trans  drv     cty   hwy fl
       class
  <chr>        <chr>      <dbl> <int> <int> <chr>  <chr> <int> <int> <
   chr> <chr>
1 honda        civic        1.6  1999     4 manua~ f        23    29 p
       subc~
2 honda        civic        1.6  1999     4 auto(~ f        24    32 r
       subc~
3 honda        civic        1.6  1999     4 auto(~ f        24    32 r
       subc~
4 honda        civic        1.6  1999     4 manua~ f        25    32 r
       subc~
5 honda        civic        1.6  1999     4 manua~ f        28    33 r
       subc~
6 audi         a4 quattro   1.8  1999     4 auto(~ 4        16    25 p
       comp~
```

- Arrange rows by value in descending order

```
mpg |>
  arrange(desc(displ)) |>
  head()
```

```
# A tibble: 6 x 11
  manufacturer model       displ  year   cyl trans drv     cty   hwy fl
       class
  <chr>        <chr>       <dbl> <int> <int> <chr> <chr> <int> <int> <
   chr> <chr>
1 chevrolet    corvette      7    2008     8 manu~ r        15    24 p
       2sea~
2 chevrolet    k1500 taho~   6.5  1999     8 auto~ 4        14    17 d
       suv
3 chevrolet    corvette      6.2  2008     8 manu~ r        16    26 p
       2sea~
4 chevrolet    corvette      6.2  2008     8 auto~ r        15    25 p
       2sea~
5 jeep         grand cher~   6.1  2008     8 auto~ 4        11    14 p
       suv
6 chevrolet    c1500 subu~   6    2008     8 auto~ r        12    17 r
       suv
```

You can use `desc()` for tie-breaker columns as well

```
mpg |>
  arrange(desc(displ), desc(cty)) |>
  head()
```

```
# A tibble: 6 x 11
  manufacturer model       displ  year   cyl trans drv     cty   hwy fl
       class
  <chr>        <chr>       <dbl> <int> <int> <chr> <chr> <int> <int> <
   chr> <chr>
1 chevrolet    corvette      7    2008     8 manu~ r        15    24 p
       2sea~
2 chevrolet    k1500 taho~   6.5  1999     8 auto~ 4        14    17 d
       suv
3 chevrolet    corvette      6.2  2008     8 manu~ r        16    26 p
       2sea~
4 chevrolet    corvette      6.2  2008     8 auto~ r        15    25 p
       2sea~
5 jeep         grand cher~   6.1  2008     8 auto~ 4        11    14 p
       suv
6 chevrolet    c1500 subu~   6    2008     8 auto~ r        12    17 r
       suv
```

- Filter rows with a logical test

```
mpg |>
  filter(model == 'jetta') |>
  head(3)
```

```
# A tibble: 3 x 11
  manufacturer model displ  year   cyl trans      drv     cty   hwy fl
      class
  <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <
   chr> <chr>
1 volkswagen   jetta   1.9  1999     4 manual(m5) f        33    44 d
      compa~
2 volkswagen   jetta   2    1999     4 manual(m5) f        21    29 r
      compa~
3 volkswagen   jetta   2    1999     4 auto(l4)   f        19    26 r
      compa~
```

- FIlter rows with more than one logical test

```
mpg |>
  filter(model=='jetta', year==1999) |>
  head(3)
```

```
# A tibble: 3 x 11
  manufacturer model displ  year   cyl trans      drv     cty   hwy fl
      class
  <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <
   chr> <chr>
1 volkswagen   jetta   1.9  1999     4 manual(m5) f        33    44 d
      compa~
2 volkswagen   jetta   2    1999     4 manual(m5) f        21    29 r
      compa~
3 volkswagen   jetta   2    1999     4 auto(l4)   f        19    26 r
      compa~
```

- Select columns by name

```
table1 |>
  select(country, year, cases) |>
  head(3)
```

```
# A tibble: 3 x 3
  country      year cases
  <chr>       <dbl> <dbl>
1 Afghanistan  1999   745
2 Afghanistan  2000  2666
3 Brazil       1999 37737
```

- Drop columns by name

```
table1 |>
  select(-c(population, year)) |>
  head(3)
```

```
# A tibble: 3 x 2
  country     cases
  <chr>       <dbl>
1 Afghanistan   745
2 Afghanistan  2666
3 Brazil      37737
```

- Select a range of columns

```
table1 |>
  select(country:cases) |>
  head(3)
```

```
# A tibble: 3 x 3
  country      year cases
  <chr>       <dbl> <dbl>
1 Afghanistan  1999   745
2 Afghanistan  2000  2666
3 Brazil       1999 37737
```

- Select columns by integer position

```
table1 |>
  select(1, 2, 4) |>
  head(3)
```

```
# A tibble: 3 x 3
  country      year population
  <chr>       <dbl>      <dbl>
1 Afghanistan  1999   19987071
2 Afghanistan  2000   20595360
3 Brazil       1999  172006362
```

- Select columns by start of name

```
table1 |>
  select(starts_with('c')) |>
  head(3)
```

```
# A tibble: 3 x 2
  country     cases
  <chr>       <dbl>
1 Afghanistan   745
2 Afghanistan  2666
3 Brazil      37737
```

- Select columns by end of name

```
table1 |>
  select(ends_with('tion')) |>
  head(3)
```

```
# A tibble: 3 x 1
  population
       <dbl>
1   19987071
2   20595360
3  172006362
```

- Select columns by string in name; to return every column whose name contains a specific string or regular expression

```
table1 |>
  select(matches('o.*u')) |>
  head(3)
```

```
# A tibble: 3 x 2
  country     population
  <chr>            <dbl>
1 Afghanistan   19987071
2 Afghanistan   20595360
3 Brazil       172006362
```

- Reorder columns

```
table1 |>
  select(country, year, population, cases) |>
  head(3)
```

```
# A tibble: 3 x 4
  country      year population cases
  <chr>       <dbl>      <dbl> <dbl>
1 Afghanistan  1999   19987071   745
2 Afghanistan  2000   20595360  2666
3 Brazil       1999  172006362 37737
```

- Reorder specific columns and leave the rest to order anyhow

```
table1 |>
  select(country, year, everything()) |>
  head(3)
```

```
# A tibble: 3 x 4
  country      year cases population
  <chr>       <dbl> <dbl>      <dbl>
1 Afghanistan  1999   745   19987071
2 Afghanistan  2000  2666   20595360
3 Brazil       1999 37737  172006362
```

- Rename Columns

```
table1 |>
  rename(state=country, date=year) |> # new name=old name
  head(3)
```

```
# A tibble: 3 x 4
  state          date cases population
  <chr>         <dbl> <dbl>      <dbl>
1 Afghanistan    1999   745   19987071
2 Afghanistan    2000  2666   20595360
3 Brazil         1999 37737  172006362
```

- Return the contents of a column as a vector

```
table1 |>
  pull(cases)
```

```
[1]    745   2666  37737  80488 212258 213766
```

You can also pull integer position

```
table1 |>
  pull(3)
```

```
[1]    745   2666  37737  80488 212258 213766
```

- Mutate data (Add new variables)

```
table1 |>
  mutate(rate = cases/population, percentage = rate*100) |>
  head(3)
```

```
# A tibble: 3 x 6
  country      year cases population      rate percentage
  <chr>       <dbl> <dbl>      <dbl>     <dbl>      <dbl>
1 Afghanistan  1999   745   19987071 0.0000373    0.00373
2 Afghanistan  2000  2666   20595360 0.000129     0.0129
3 Brazil       1999 37737  172006362 0.000219     0.0219
```

- **Dropping the original data**; to return only new columns that `mutate()` would create

```
table1 |>
  transmute(rate = cases/population, percentage = rate*100) |>
  head(3)
```

```
# A tibble: 3 x 2
       rate percentage
      <dbl>      <dbl>
1 0.0000373    0.00373
2 0.000129     0.0129
3 0.000219     0.0219
```

- **Summarise data**; to compute summary statistics

```
table1 |>
  summarise(total_cases = sum(cases), max_rate = max(cases/population))
```

```
# A tibble: 1 x 2
  total_cases max_rate
        <dbl>    <dbl>
1      547660 0.000461
```

- Group data

```
table1 |>
  group_by(country)
```

```
# A tibble: 6 x 4
# Groups:   country [3]
  country      year  cases population
  <chr>       <dbl>  <dbl>      <dbl>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

- Summarise data by groups

```
table1 |>
  group_by(country) |>
  summarise(total_cases = sum(cases), max_rate = max(cases/population))
```

```
# A tibble: 3 x 3
  country     total_cases max_rate
  <chr>             <dbl>    <dbl>
1 Afghanistan        3411 0.000129
2 Brazil           118225 0.000461
3 China            426024 0.000167
```

- **Nest a data frame**; to move portions of your data frame into their own tables, and then store those tables in cells in your original data frame.

```
nested_iris <- iris |>
  group_by(Species) |>
  nest(.key='Measurements') |> # key for providing name for new list
    column
  as_tibble()

nested_iris
```

```
# A tibble: 3 x 2
  Species    Measurements
  <fct>      <list>
1 setosa     <tibble [50 x 4]>
2 versicolor <tibble [50 x 4]>
3 virginica  <tibble [50 x 4]>
```

`nest()` preserves class, which means that `nest()` will return a data frame if its input is a data frame and a tibble if its input is a tibble so it is recommend that you convert the result of `nest()` to a tibble when necessary.

- Extract a table from a nested dataframe

```
nested_iris |>
  filter(Species == 'setosa') |>
  unnest(cols=Measurements)
```

```
# A tibble: 50 x 5
   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
   <fct>          <dbl>       <dbl>        <dbl>       <dbl>
```

```
 1 setosa          5.1        3.5        1.4        0.2
 2 setosa          4.9        3          1.4        0.2
 3 setosa          4.7        3.2        1.3        0.2
 4 setosa          4.6        3.1        1.5        0.2
 5 setosa          5          3.6        1.4        0.2
 6 setosa          5.4        3.9        1.7        0.4
 7 setosa          4.6        3.4        1.4        0.3
 8 setosa          5          3.4        1.5        0.2
 9 setosa          4.4        2.9        1.4        0.2
10 setosa          4.9        3.1        1.5        0.1
# i 40 more rows
```

- Unnest a dataframe

```
nested_iris |> unnest(cols=Measurements) |> head(3)
```

```
# A tibble: 3 x 5
  Species Sepal.Length Sepal.Width Petal.Length Petal.Width
  <fct>          <dbl>       <dbl>        <dbl>       <dbl>
1 setosa           5.1         3.5          1.4         0.2
2 setosa           4.9         3            1.4         0.2
3 setosa           4.7         3.2          1.3         0.2
```

- Join datasets by common column(s)

For example, you would like to combine `band_members` and `band_instruments` into a single data frame based on the values of the `name` column.

```
print(band_members)
```

```
# A tibble: 3 x 2
  name  band
  <chr> <chr>
1 Mick  Stones
2 John  Beatles
3 Paul  Beatles
```

```
print(band_instruments)
```

```
# A tibble: 3 x 2
  name  plays
  <chr> <chr>
1 John  guitar
2 Paul  bass
3 Keith guitar
```

```
band_members |>
  left_join(band_instruments, by='name')
```

```
# A tibble: 3 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 Mick  Stones  <NA>
2 John  Beatles guitar
3 Paul  Beatles bass
```

There are four ways to join contenct from one data frame to another:

`left_join()` drops any row in the *second* data set does not match a row in the first data set.

`right_join()` drops any row in the *first* data set does not match a row in the first data set.

`inner_join()` drops any row in *either* data set that does not have a match in both data sets

`full_join()` retains every row from both data sets; it is the only join guaranteed to retain all of the original data.

- Specifying column(s) to join on

```
table1 |>
  left_join(table3, by=c('country', 'year'))
```

```
# A tibble: 6 x 5
  country      year  cases population rate
  <chr>       <dbl>  <dbl>      <dbl> <chr>
1 Afghanistan  1999    745   19987071 745/19987071
2 Afghanistan  2000   2666   20595360 2666/20595360
3 Brazil       1999  37737  172006362 37737/172006362
4 Brazil       2000  80488  174504898 80488/174504898
5 China        1999 212258 1272915272 212258/1272915272
6 China        2000 213766 1280428583 213766/1280428583
```

- Find rows that have a match in another data set

```
band_members |>
  semi_join(band_instruments, by='name') # returns only the rows of the
    first df that have a match(refering to same observation even if
   diff measurements) in the second df
```

```
# A tibble: 2 x 2
  name  band
  <chr> <chr>
1 John  Beatles
2 Paul  Beatles
```

- Find rows that do not have a match in another data set

```
band_members |>
  anti_join(band_instruments, by='name') # returns only the rows of the
    first data frame that do not have a match in the second data frame
```

```
# A tibble: 1 x 2
  name  band
  <chr> <chr>
1 Mick  Stones
```

5 Transform Lists and Vectors

- Extract an element from a list

```
state.center |> pluck('x') # returns element named x in state.center
```

```
 [1]  -86.7509 -127.2500 -111.6250  -92.2992 -119.7730 -105.5130
   -72.3573
 [8]  -74.9841  -81.6850  -83.3736 -126.2500 -113.9300  -89.3776
   -86.0808
[15]  -93.3714  -98.1156  -84.7674  -92.2724  -68.9801  -76.6459
   -71.5800
[22]  -84.6870  -94.6043  -89.8065  -92.5137 -109.3200  -99.5898
   -116.8510
[29]  -71.3924  -74.2336 -105.9420  -75.1449  -78.4686 -100.0990
   -82.5963
[36]  -97.1239 -120.0680  -77.4500  -71.1244  -80.5056  -99.7238
   -86.4560
[43]  -98.7857 -111.3300  -72.5450  -78.2005 -119.7460  -80.6665
   -89.9941
[50] -107.2560
```