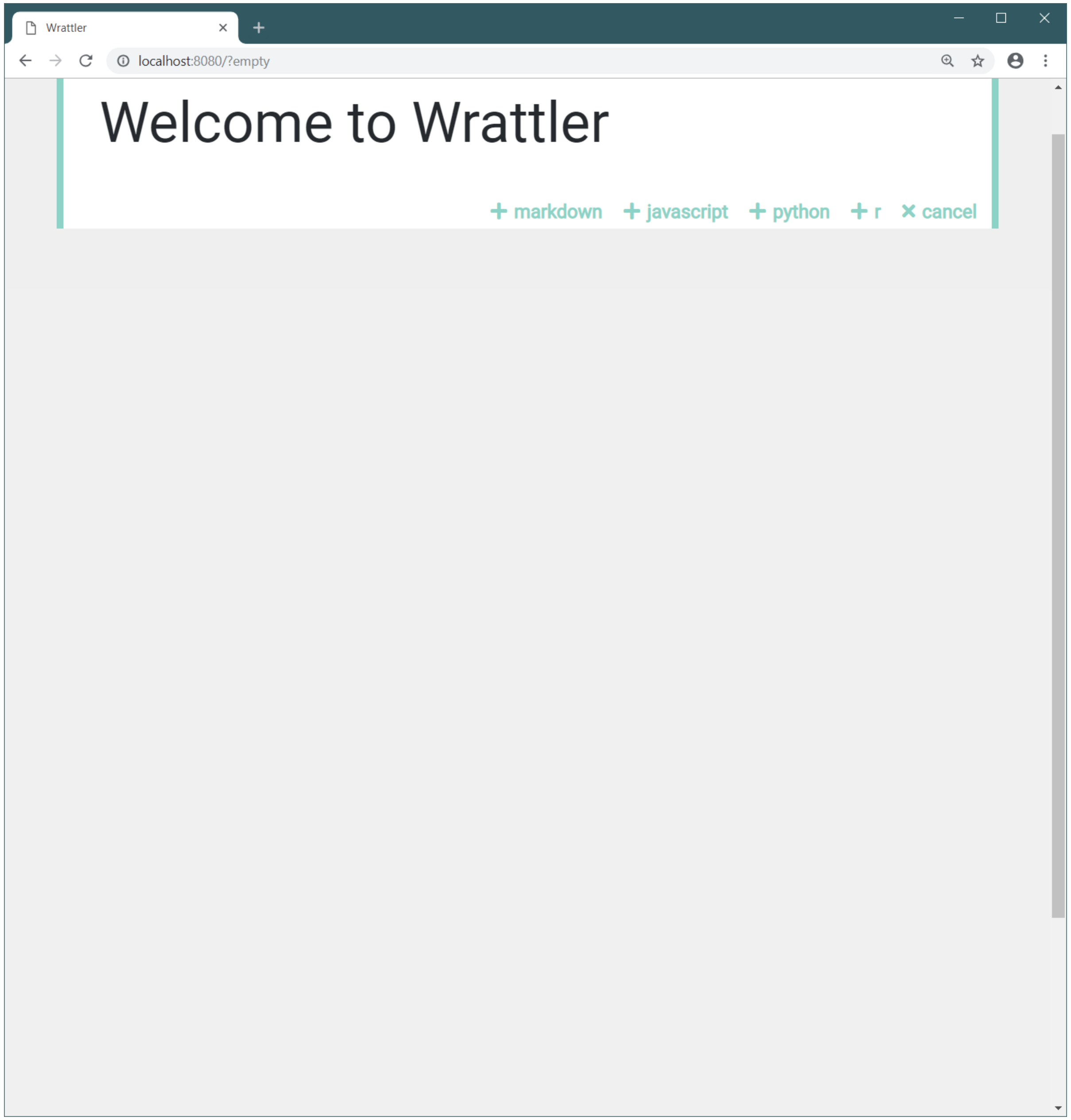
Wrattler: Polyglot, reproducible and smart notebooks

Demonstration of the developed prototype system (GDI 1.6b)

Polyglot

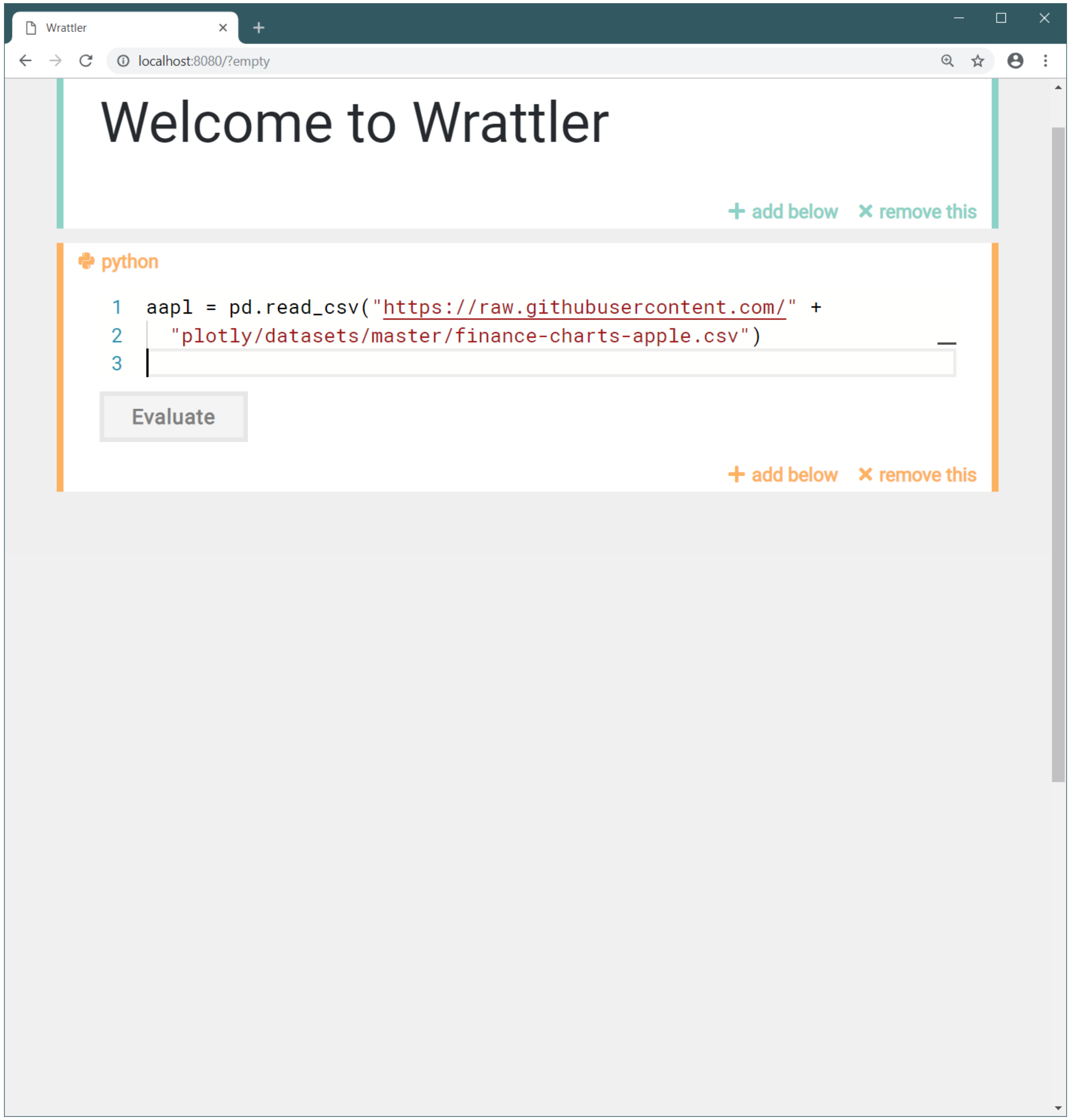
Passing data from Python to JavaScript


Wrattler

localhost:8080/?empty

Welcome to Wrattler

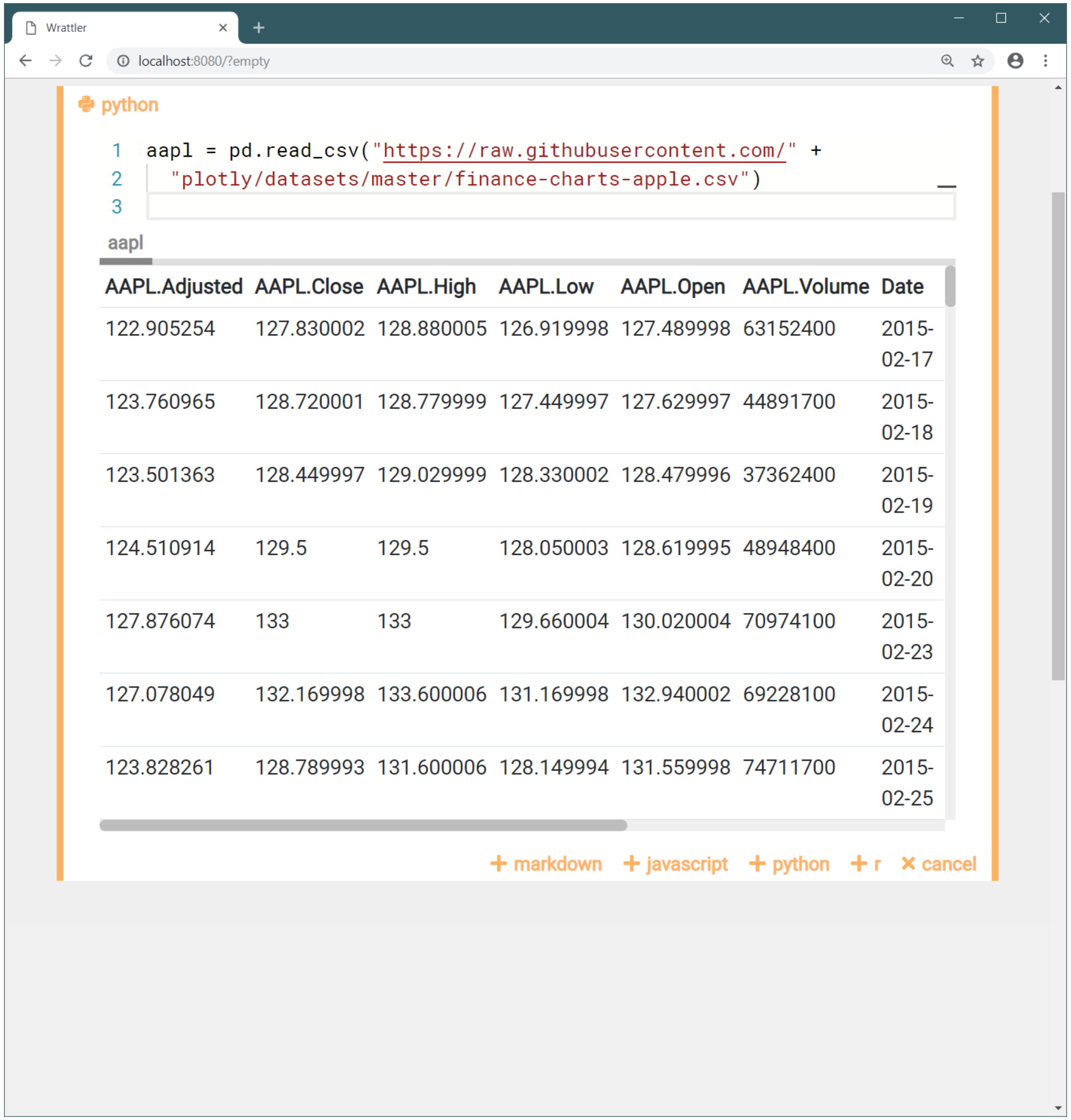
We start with an empty notebook. Wrattler allows us to add with **Markdown** comments and cells with **JavaScript, R** and **Python** code.



Welcome to Wrattler

```
aapl = pd.read_csv("https://raw.githubusercontent.com/" +
  "plotly/datasets/master/finance-charts-apple.csv")

```

We use the Python **pandas** library to easily load CSV file from an online source.

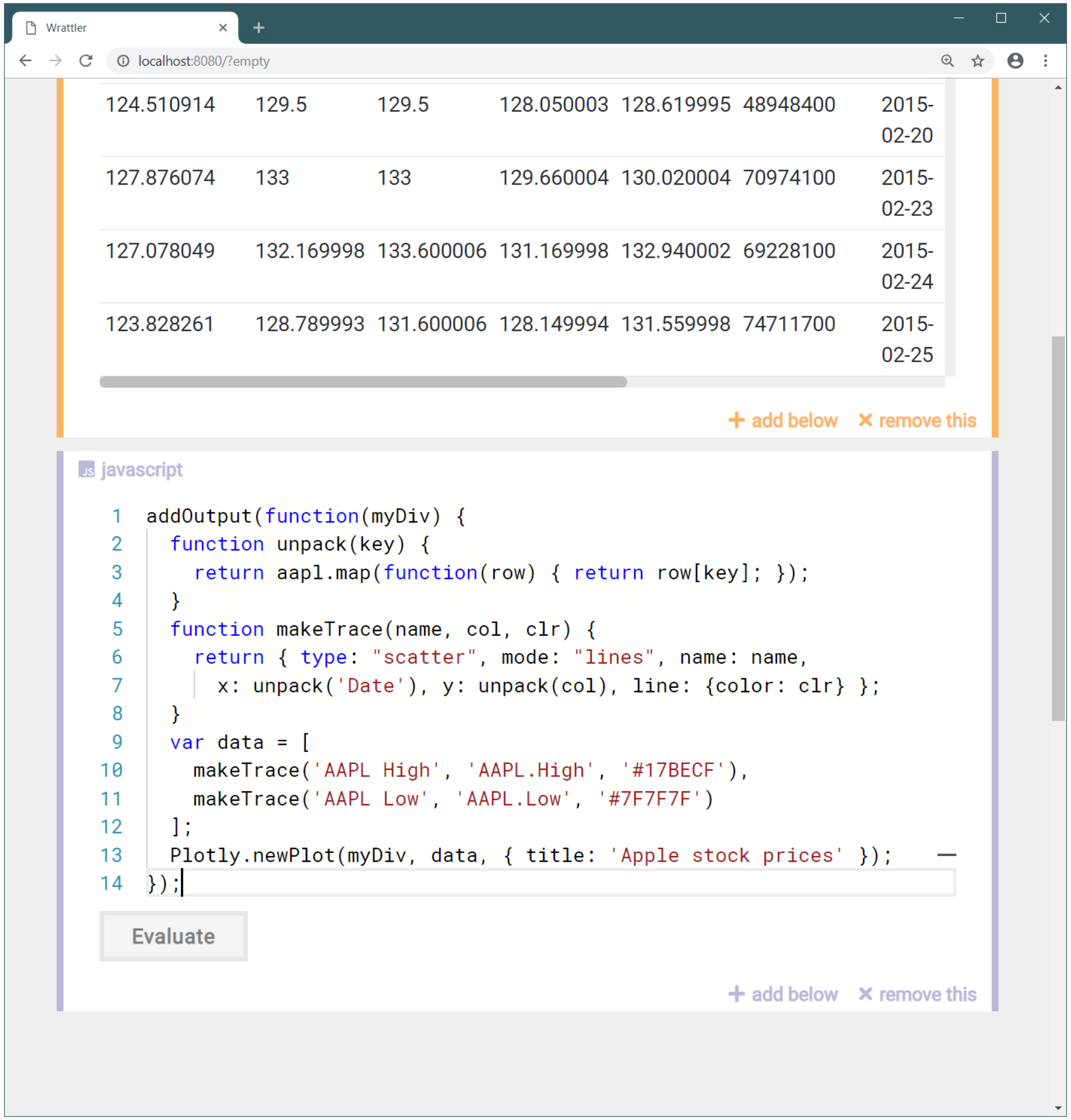


```
aapl = pd.read_csv("https://raw.githubusercontent.com/" +
  "plotly/datasets/master/finance-charts-apple.csv")
```

aapl

AAPL.Adjusted	AAPL.Close	AAPL.High	AAPL.Low	AAPL.Open	AAPL.Volume	Date
122.905254	127.830002	128.880005	126.919998	127.489998	63152400	2015-02-17
123.760965	128.720001	128.779999	127.449997	127.629997	44891700	2015-02-18
123.501363	128.449997	129.029999	128.330002	128.479996	37362400	2015-02-19
124.510914	129.5	129.5	128.050003	128.619995	48948400	2015-02-20
127.876074	133	133	129.660004	130.020004	70974100	2015-02-23
127.078049	132.169998	133.600006	131.169998	132.940002	69228100	2015-02-24
123.828261	128.789993	131.600006	128.149994	131.559998	74711700	2015-02-25

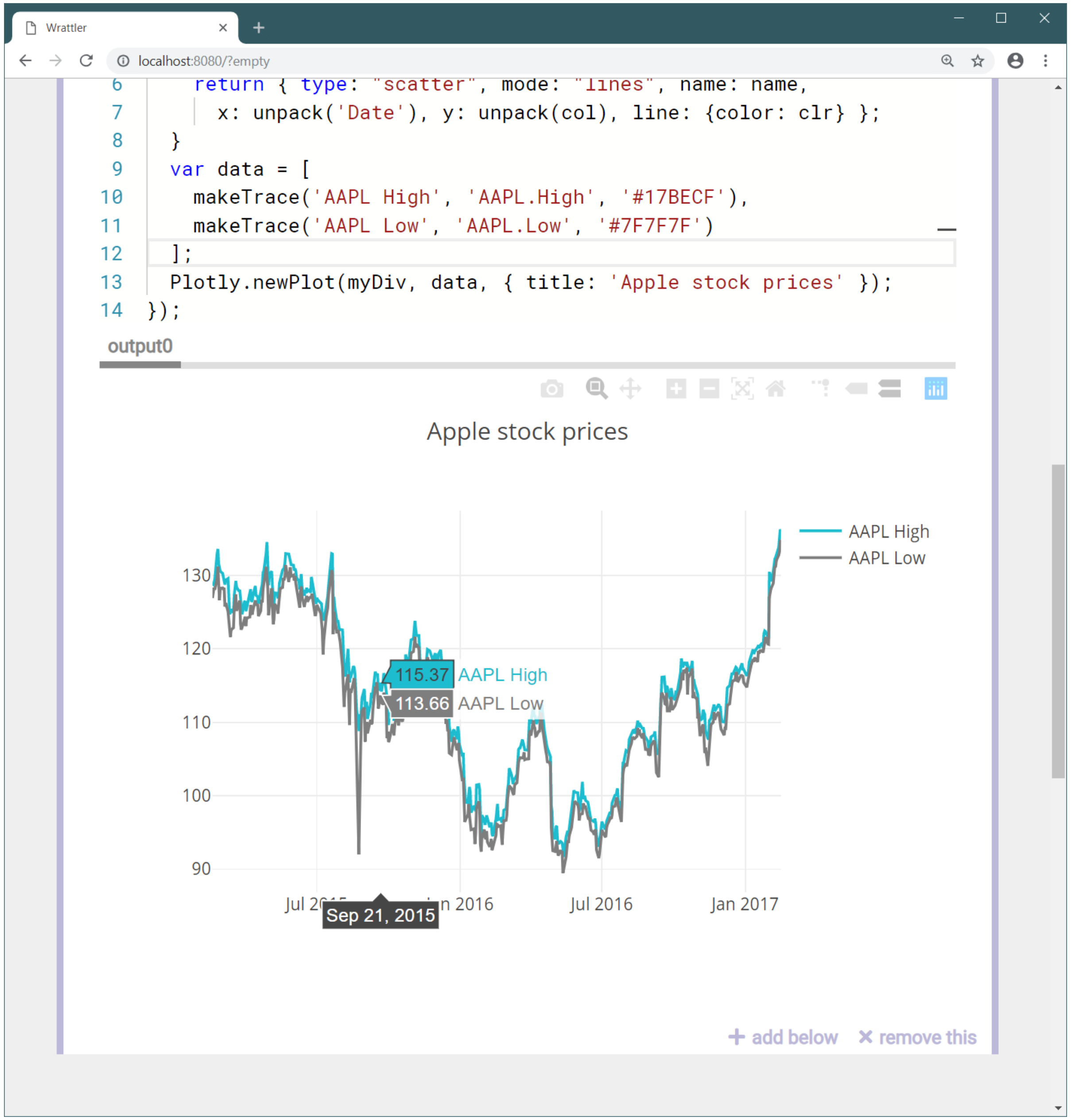
After updating the cell and clicking the **Evaluate** button, we see a preview of the downloaded data.



124.510914	129.5	129.5	128.050003	128.619995	48948400	2015-02-20
127.876074	133	133	129.660004	130.020004	70974100	2015-02-23
127.078049	132.169998	133.600006	131.169998	132.940002	69228100	2015-02-24
123.828261	128.789993	131.600006	128.149994	131.559998	74711700	2015-02-25

```
addOutput(function(myDiv) {
  function unpack(key) {
    return aapl.map(function(row) { return row[key]; });
  }
  function makeTrace(name, col, clr) {
    return { type: "scatter", mode: "lines", name: name,
      x: unpack('Date'), y: unpack(col), line: {color: clr} };
  }
  var data = [
    makeTrace('AAPL High', 'AAPL.High', '#17BECF'),
    makeTrace('AAPL Low', 'AAPL.Low', '#7F7F7F')
  ];
  Plotly.newPlot(myDiv, data, { title: 'Apple stock prices' });
});
```

Next, we add a **JavaScript** cell with source code that uses the **Plotly** library to build a visualization.

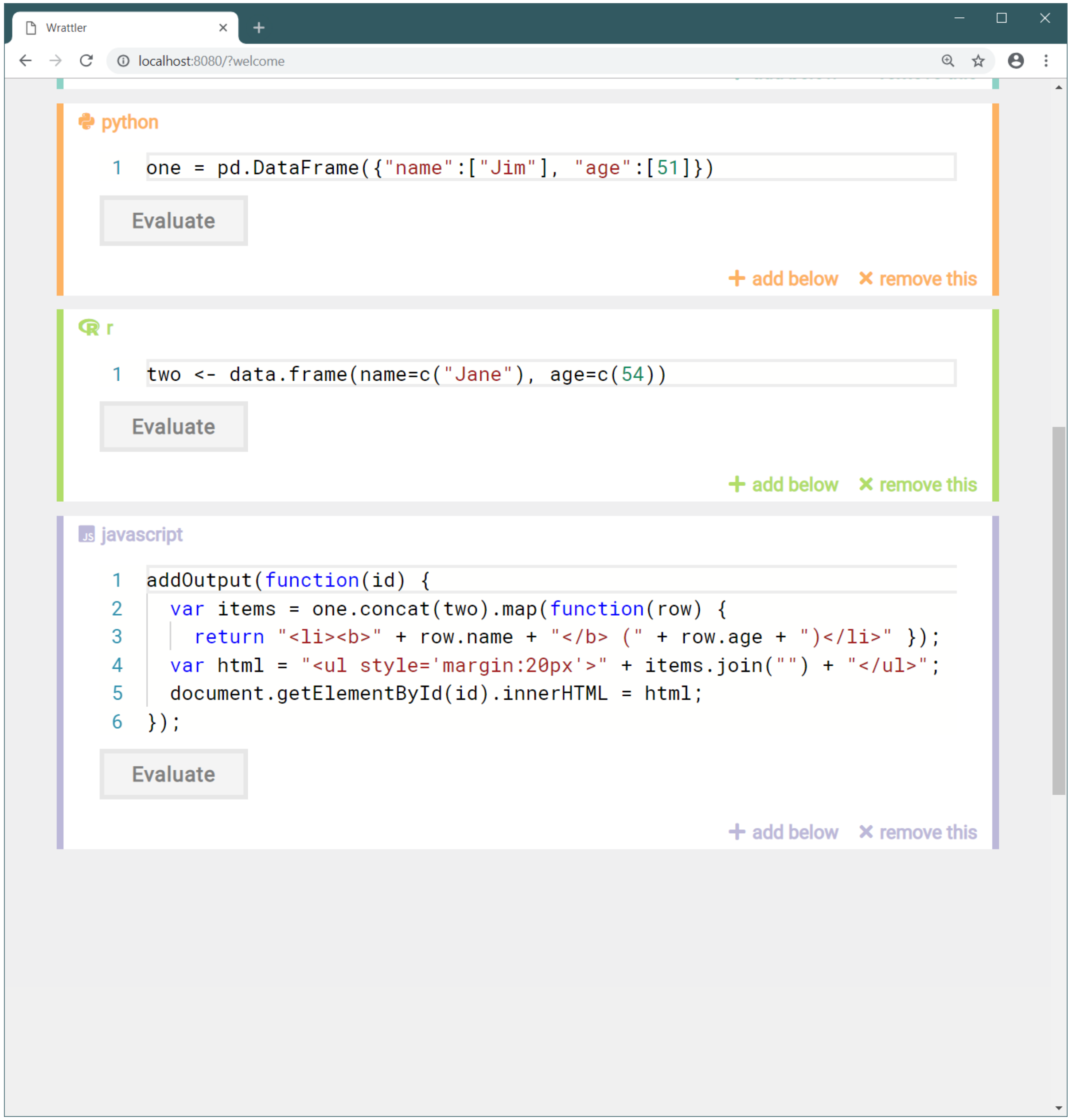


```
    return { type: "scatter", mode: "lines", name: name,
      x: unpack('Date'), y: unpack(col), line: {color: clr} };
  }
  var data = [
    makeTrace('AAPL High', 'AAPL.High', '#17BECF'),
    makeTrace('AAPL Low', 'AAPL.Low', '#7F7F7F')
  ];
  Plotly.newPlot(myDiv, data, { title: 'Apple stock prices' });
});
```

When we evaluate the added code, Wrattler runs our **JavaScript** directly **in the web browser** and shows the visualization.

Reproducible

Recomputation using provenance

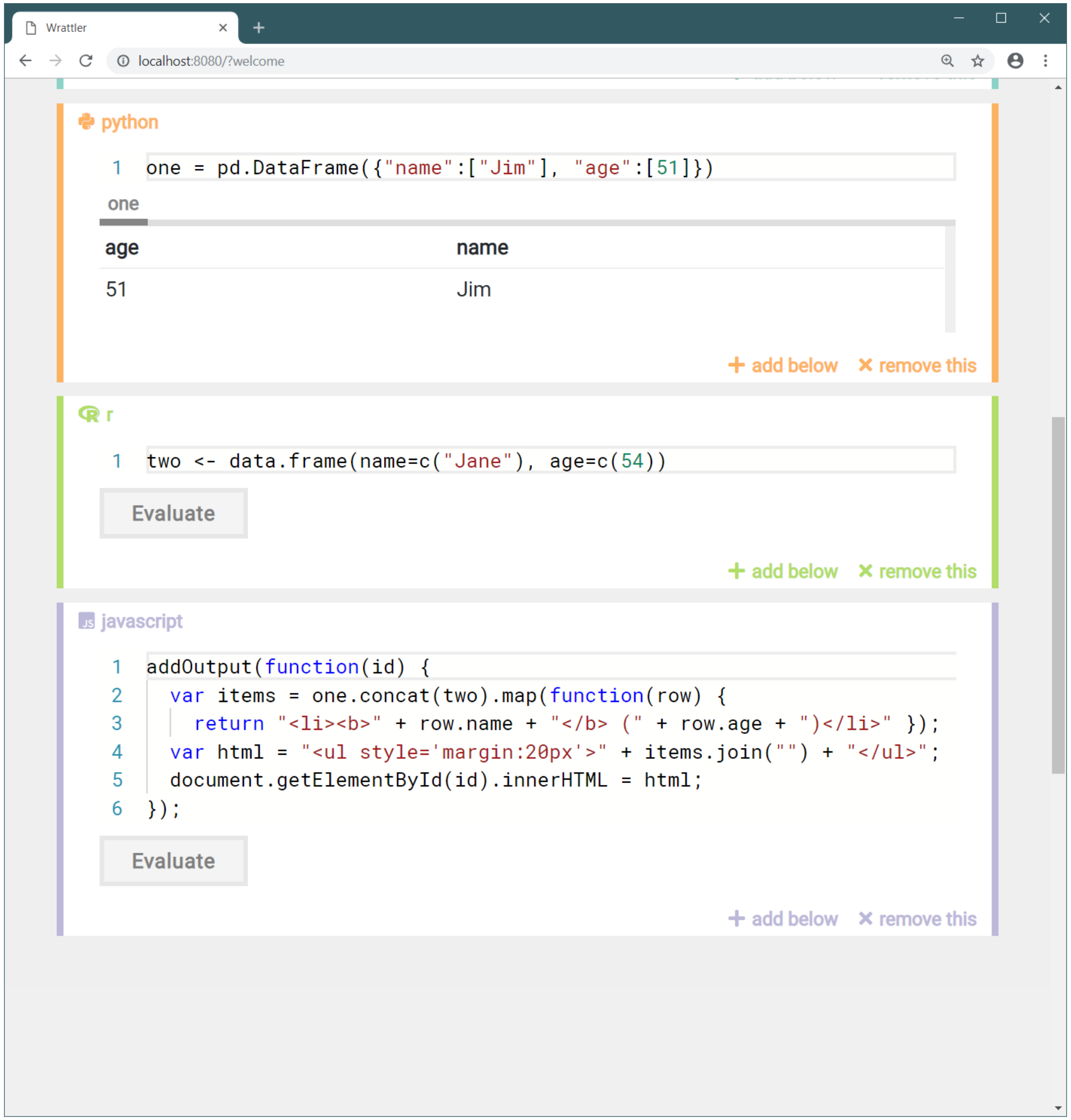


```
one = pd.DataFrame({"name":["Jim"], "age":[51]})
```

```
two <- data.frame(name=c("Jane"), age=c(54))
```

```
addOutput(function(id) {
  var items = one.concat(two).map(function(row) {
    return "<li><b>" + row.name + "</b> (" + row.age + ")</li>" });
  var html = "<ul style='margin:20px'>" + items.join("") + "</ul>";
  document.getElementById(id).innerHTML = html;
});
```

Consider a sample notebook with two cells that define data frames `one` and `two` and a third cell that concatenates the two and prints the result.



```
one = pd.DataFrame({"name":["Jim"], "age":[51]})
```

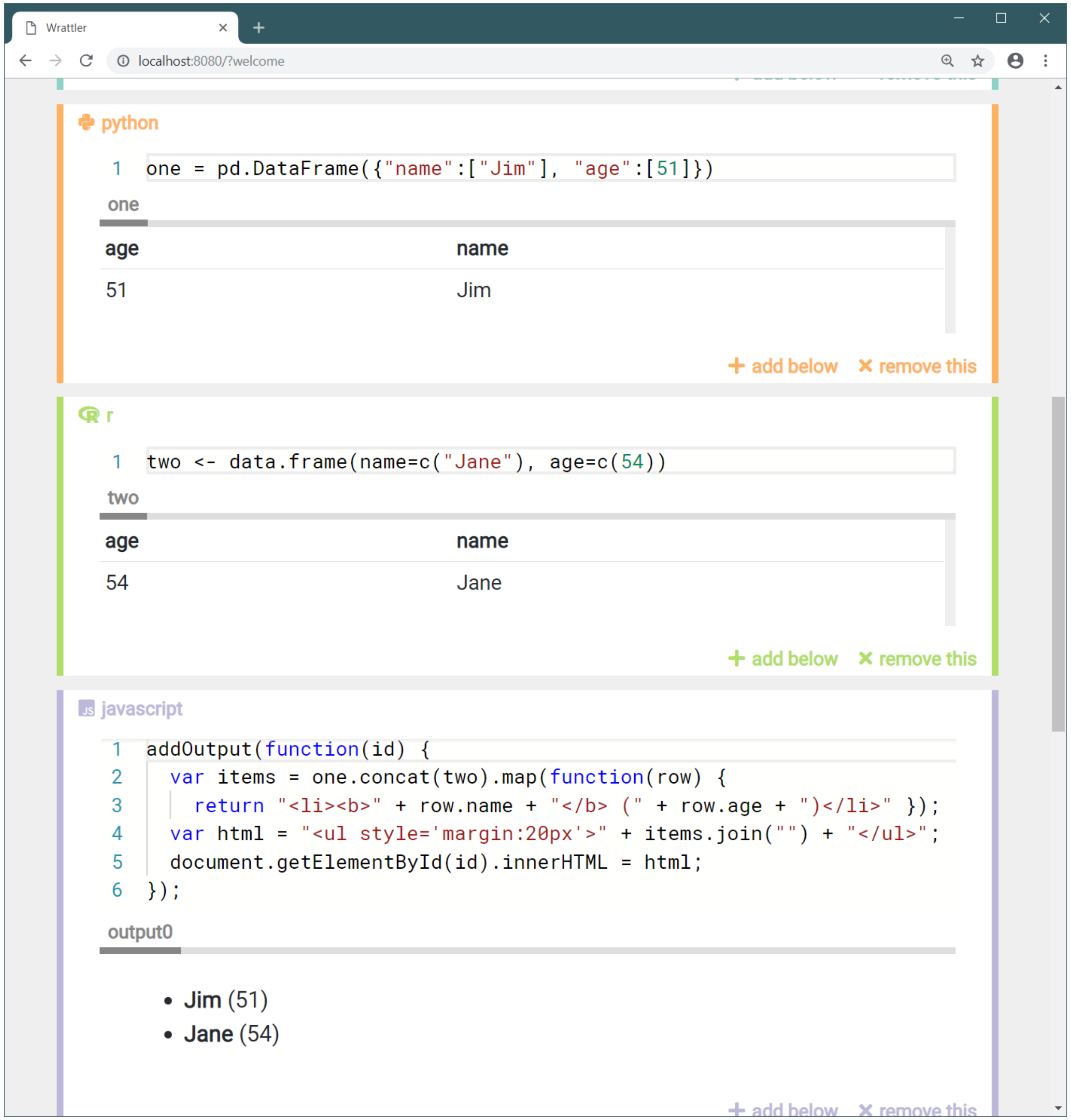
one

age	name
51	Jim

```
two <- data.frame(name=c("Jane"), age=c(54))
```

```
addOutput(function(id) {
  var items = one.concat(two).map(function(row) {
    return "<li><b>" + row.name + "</b> (" + row.age + ")</li>" });
  var html = "<ul style='margin:20px'>" + items.join("") + "</ul>";
  document.getElementById(id).innerHTML = html;
});
```

We can evaluate cells one-by-one by clicking on **Evaluate.** Here, we evaluated just the first (Python) cell.



```
one = pd.DataFrame({"name":["Jim"], "age":[51]})
```

one

age	name
51	Jim

```
two <- data.frame(name=c("Jane"), age=c(54))
```

two

age	name
54	Jane

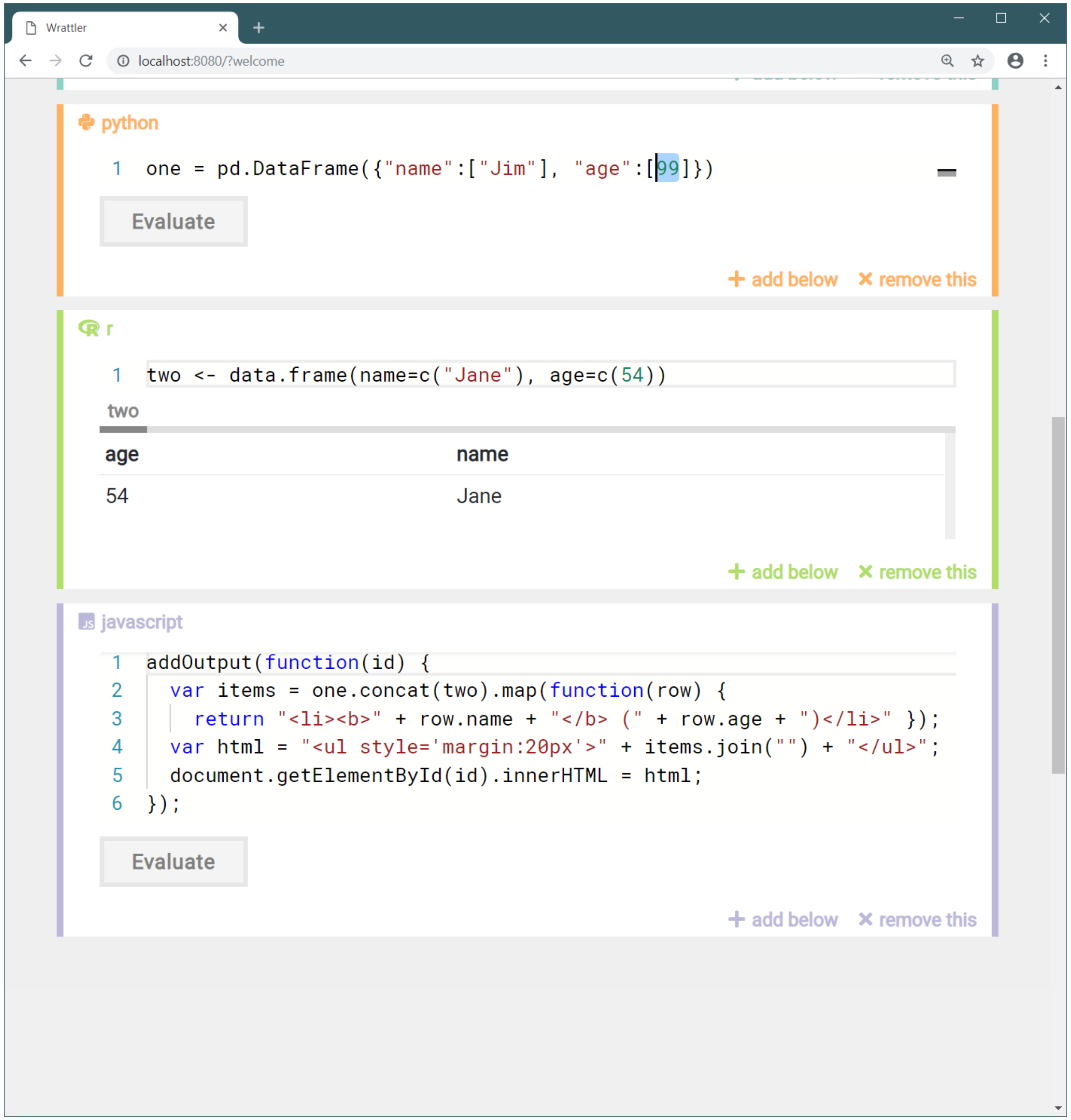
```
addOutput(function(id) {
  var items = one.concat(two).map(function(row) {
    return "<li><b>" + row.name + "</b> (" + row.age + ")</li>" });
  var html = "<ul style='margin:20px'>" + items.join("") + "</ul>";
  document.getElementById(id).innerHTML = html;
});
```

output0

- **Jim** (51)
- **Jane** (54)

But you **do not have to** run cells one-by-one. If we ask to evaluate the last cell, Wrattler automatically **runs all dependencies.**

Here, we evaluated the last cell and the second was evaluated automatically.



```
one = pd.DataFrame({"name":["Jim"], "age":[99]})
```

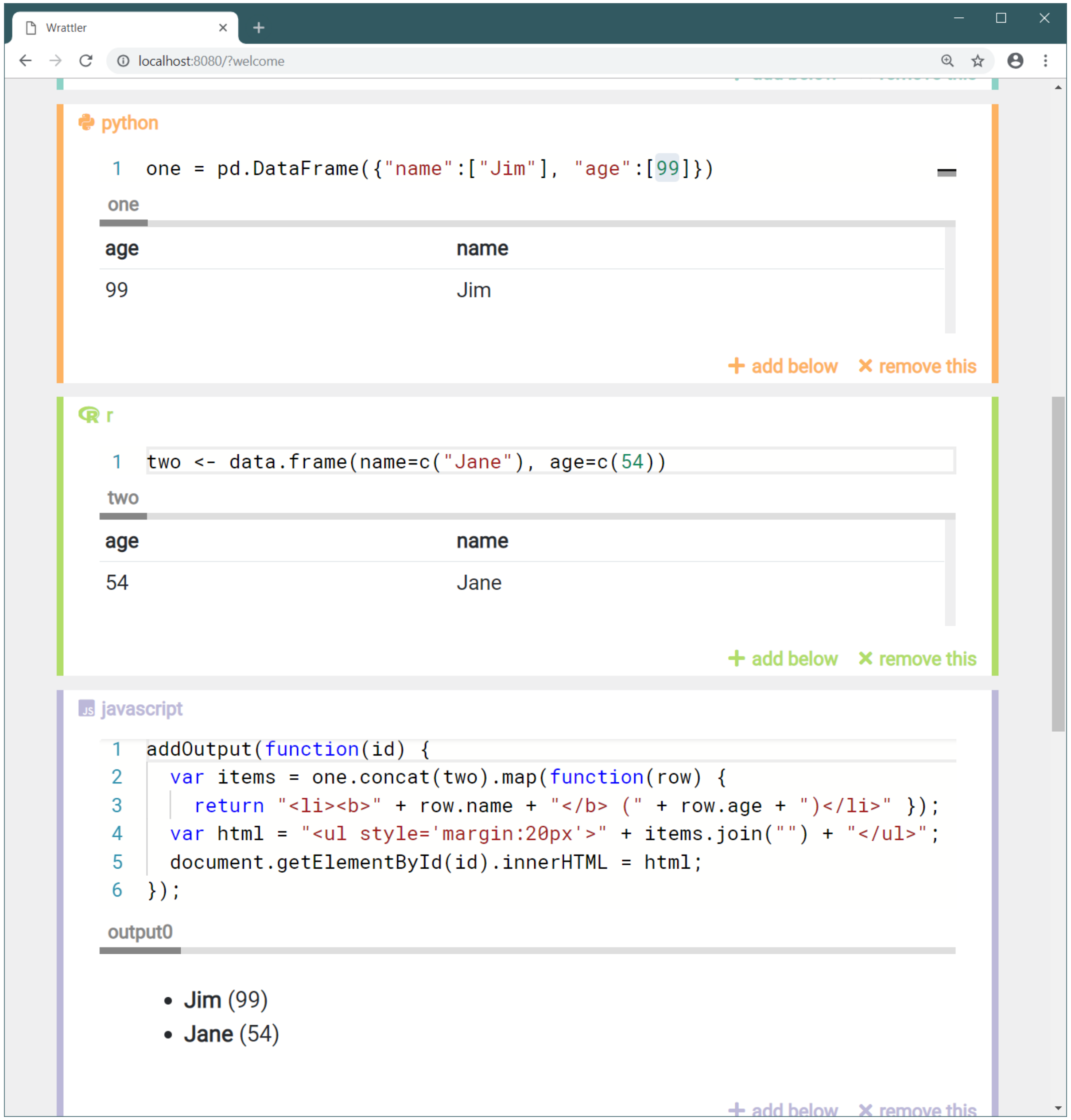
```
two <- data.frame(name=c("Jane"), age=c(54))
```

two

age	name
54	Jane

```
addOutput(function(id) {
  var items = one.concat(two).map(function(row) {
    return "<li><b>" + row.name + "</b> (" + row.age + ")</li>" });
  var html = "<ul style='margin:20px'>" + items.join("") + "</ul>";
  document.getElementById(id).innerHTML = html;
});
```

If we modify a cell, Wrattler updates the **dependency graph** it maintains. Results of all cells that depend on a modified cell are removed and need to be recomputed.



```
one = pd.DataFrame({"name":["Jim"], "age":[99]})
```

one

age	name
99	Jim

```
two <- data.frame(name=c("Jane"), age=c(54))
```

two

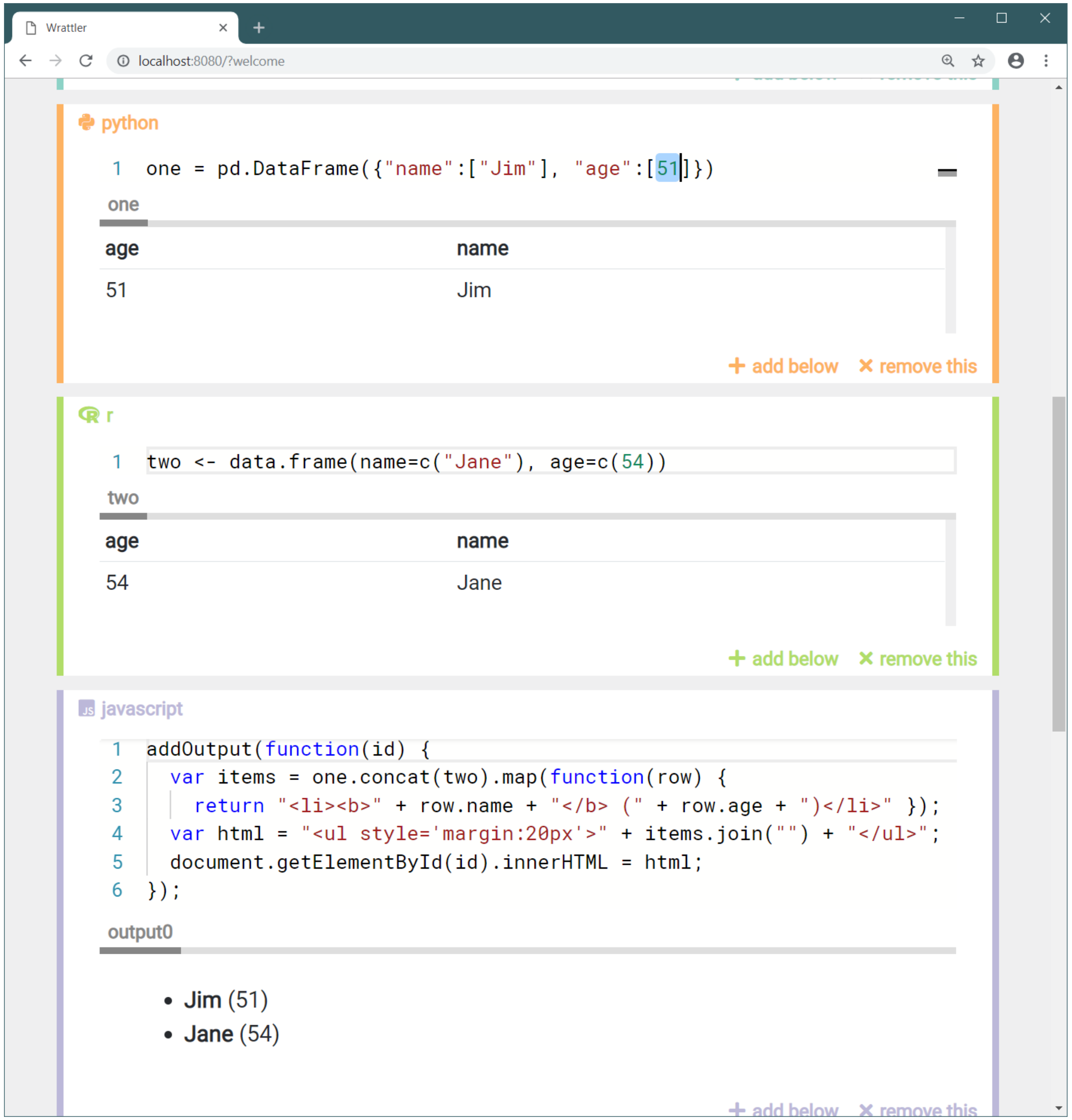
age	name
54	Jane

```
addOutput(function(id) {
  var items = one.concat(two).map(function(row) {
    return "<li><b>" + row.name + "</b> (" + row.age + ")</li>" });
  var html = "<ul style='margin:20px'>" + items.join("") + "</ul>";
  document.getElementById(id).innerHTML = html;
});
```

output0

- **Jim** (99)
- **Jane** (54)

Re-evaluating the last cell also evaluates all cells that it depends on, using the new **dependency graph.**


Wrattler
localhost:8080/?welcome

python

```
one = pd.DataFrame({"name":["Jim"], "age":[51]})
```

one

age	name
51	Jim

+ add below ✕ remove this

r

```
two <- data.frame(name=c("Jane"), age=c(54))
```

two

age	name
54	Jane

+ add below ✕ remove this

javascript

```
addOutput(function(id) {
  var items = one.concat(two).map(function(row) {
    return "<li><b>" + row.name + "</b> (" + row.age + ")</li>" });
  var html = "<ul style='margin:20px'>" + items.join("") + "</ul>";
  document.getElementById(id).innerHTML = html;
});
```

output0

- **Jim** (51)
- **Jane** (54)

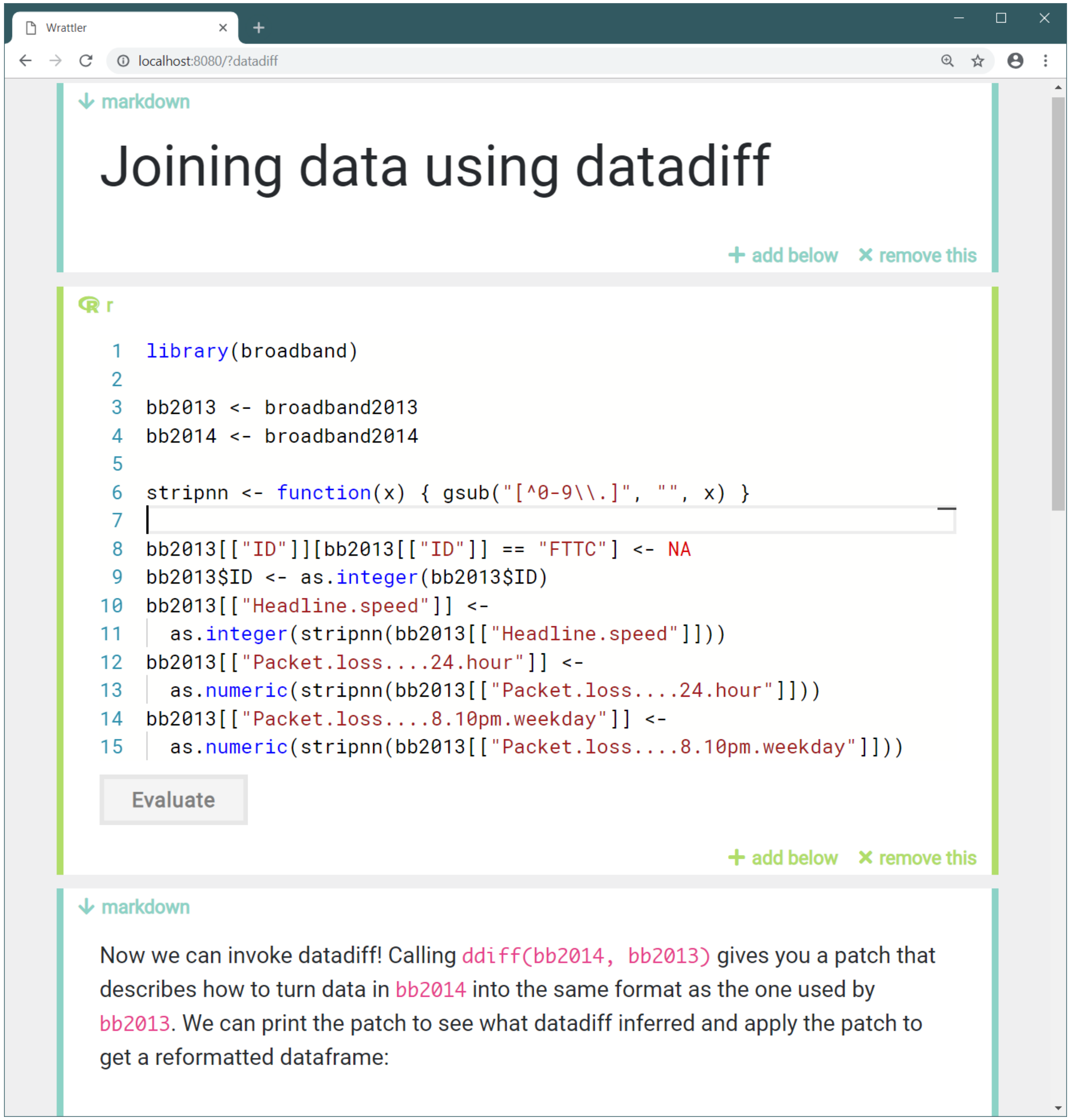
+ add below ✕ remove this

If we revert a change back, we do not have to re-evaluate and we see **earlier outputs immediately.**

Wrattler caches past dependency graph nodes and reuses a past node that has already been evaluated.

Smart

Support for data cleaning tools

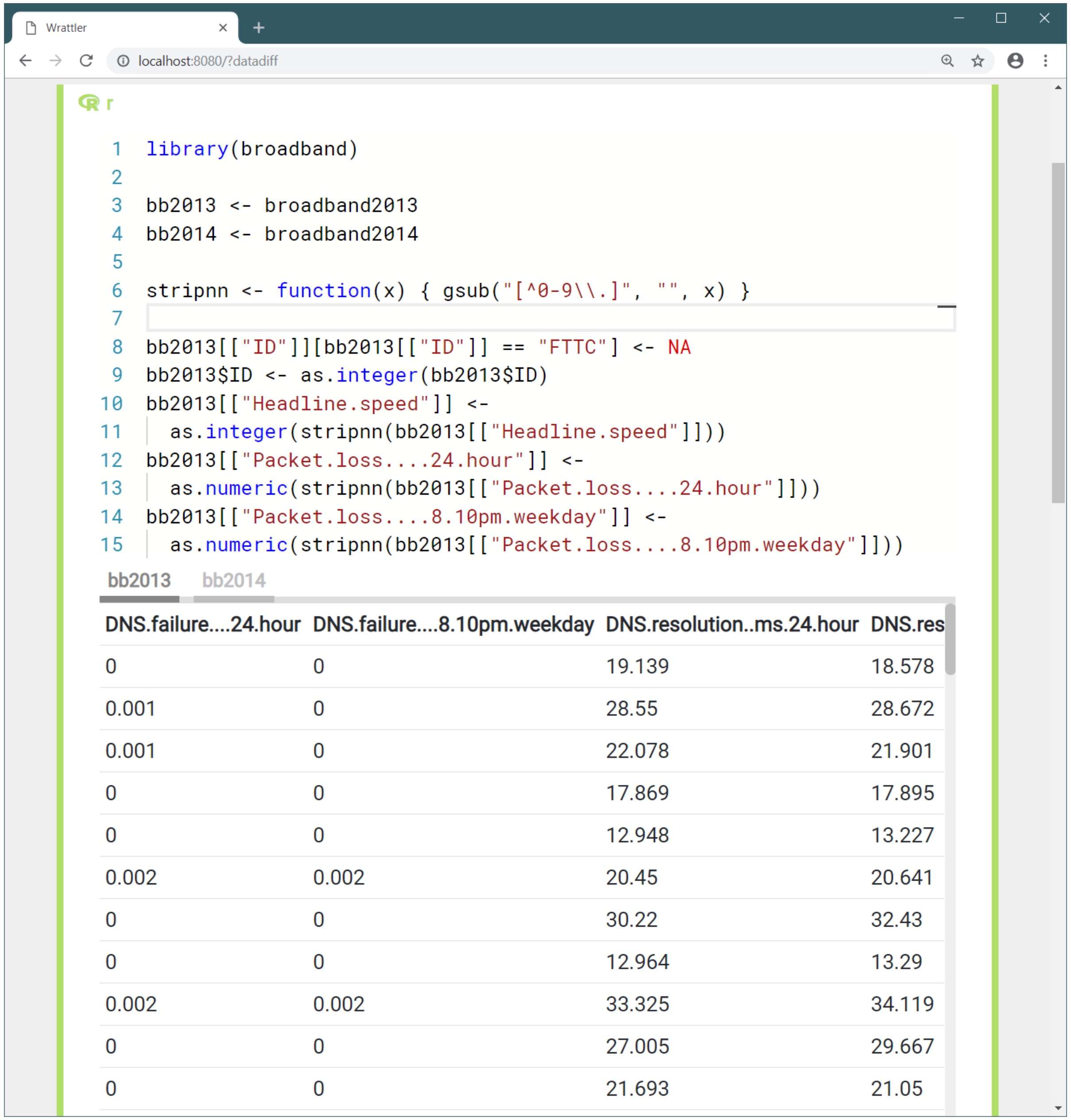


Joining data using datadiff

```r
library(broadband)

bb2013 <- broadband2013
bb2014 <- broadband2014

stripnn <- function(x) { gsub("[^0-9\\.]", "", x) }

bb2013[["ID"]][bb2013[["ID"]] == "FTTC"] <- NA
bb2013$ID <- as.integer(bb2013$ID)
bb2013[["Headline.speed"]] <-
  as.integer(stripnn(bb2013[["Headline.speed"]]))
bb2013[["Packet.loss....24.hour"]] <-
  as.numeric(stripnn(bb2013[["Packet.loss....24.hour"]]))
bb2013[["Packet.loss....8.10pm.weekday"]] <-
  as.numeric(stripnn(bb2013[["Packet.loss....8.10pm.weekday"]]))
```

Now we can invoke datadiff! Calling `ddiff(bb2014, bb2013)` gives you a patch that describes how to turn data in `bb2014` into the same format as the one used by `bb2013`. We can print the patch to see what datadiff inferred and apply the patch to get a reformatted dataframe:

Wrattler supports other data wrangling and cleaning tools built as part of the **AI for Data Analytics** project such as **datadiff.** As an example, we look at the UK broadband quality data set.


Wrattler

localhost:8080/?datadiff

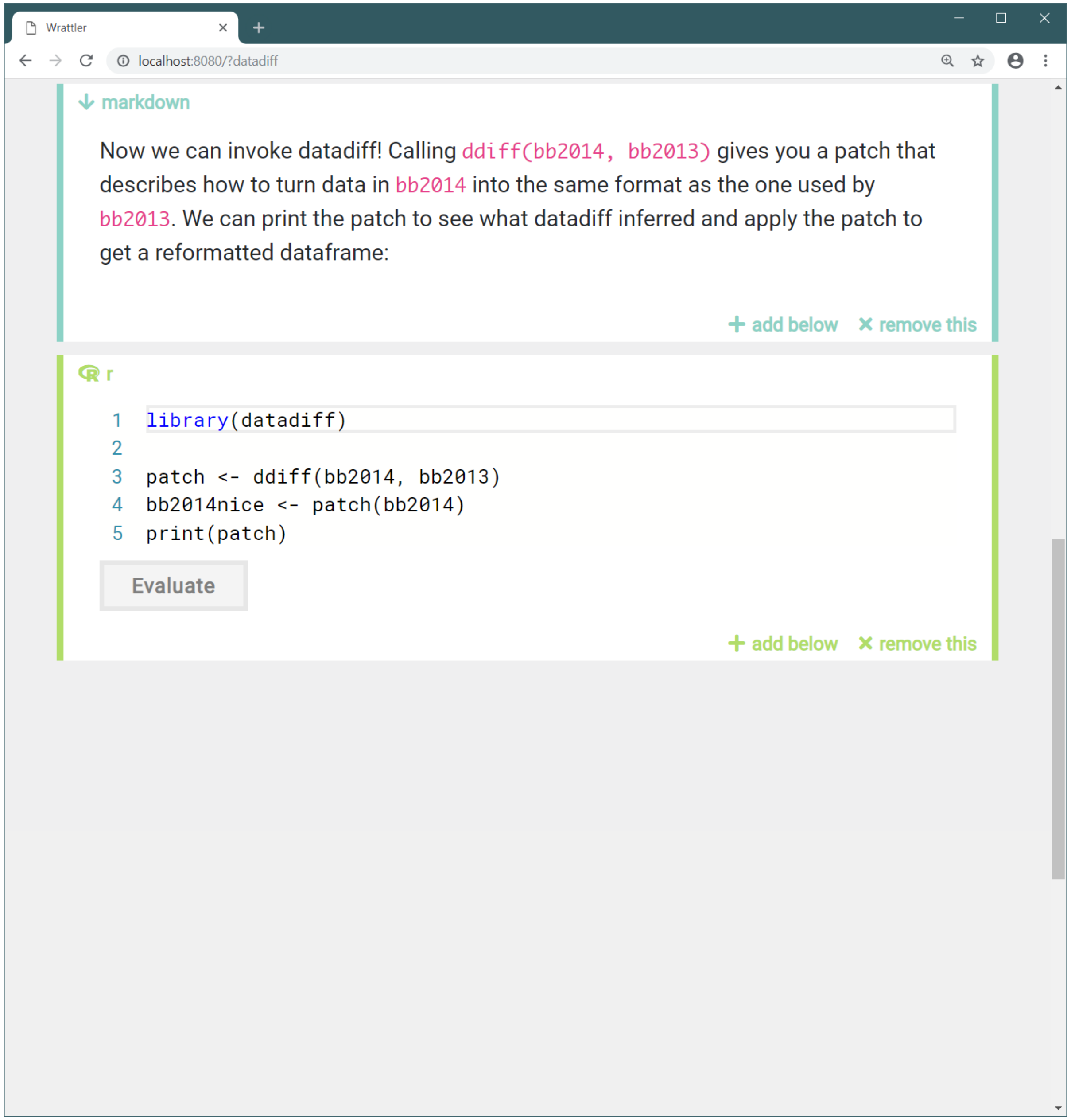
r

```
library(broadband)

bb2013 <- broadband2013
bb2014 <- broadband2014

stripnn <- function(x) { gsub("[^0-9\\.]", "", x) }

bb2013[["ID"]][bb2013[["ID"]] == "FTTC"] <- NA
bb2013$ID <- as.integer(bb2013$ID)
bb2013[["Headline.speed"]] <-
  as.integer(stripnn(bb2013[["Headline.speed"]]))
bb2013[["Packet.loss....24.hour"]] <-
  as.numeric(stripnn(bb2013[["Packet.loss....24.hour"]]))
bb2013[["Packet.loss....8.10pm.weekday"]] <-
  as.numeric(stripnn(bb2013[["Packet.loss....8.10pm.weekday"]]))
```

bb2013 bb2014

DNS.failure....24.hour	DNS.failure....8.10pm.weekday	DNS.resolution..ms.24.hour	DNS.res
0	0	19.139	18.578
0.001	0	28.55	28.672
0.001	0	22.078	21.901
0	0	17.869	17.895
0	0	12.948	13.227
0.002	0.002	20.45	20.641
0	0	30.22	32.43
0	0	12.964	13.29
0.002	0.002	33.325	34.119
0	0	27.005	29.667
0	0	21.693	21.05

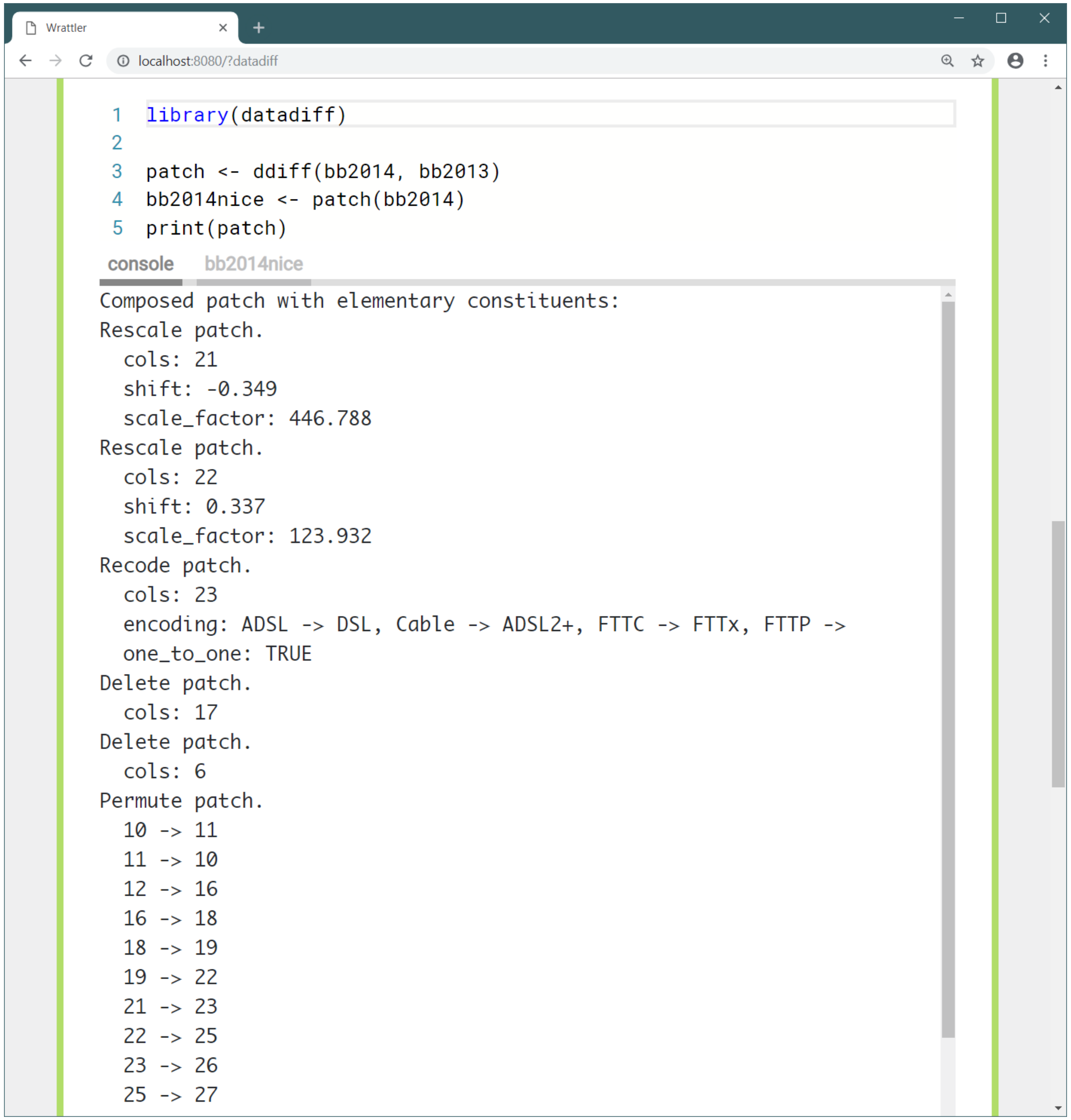
We load two data sets, **bb2013** and **bb2014** which represent same data for two years, but with some differences in the file structure.



Now we can invoke datadiff! Calling `ddiff(bb2014, bb2013)` gives you a patch that describes how to turn data in `bb2014` into the same format as the one used by `bb2013`. We can print the patch to see what datadiff inferred and apply the patch to get a reformatted dataframe:

```
library(datadiff)

patch <- ddiff(bb2014, bb2013)
bb2014nice <- patch(bb2014)
print(patch)
```

To do any data analysis, we need to reconcile the file structure. For this, we can run **datadiff.** by adding an R cell to our notebook.

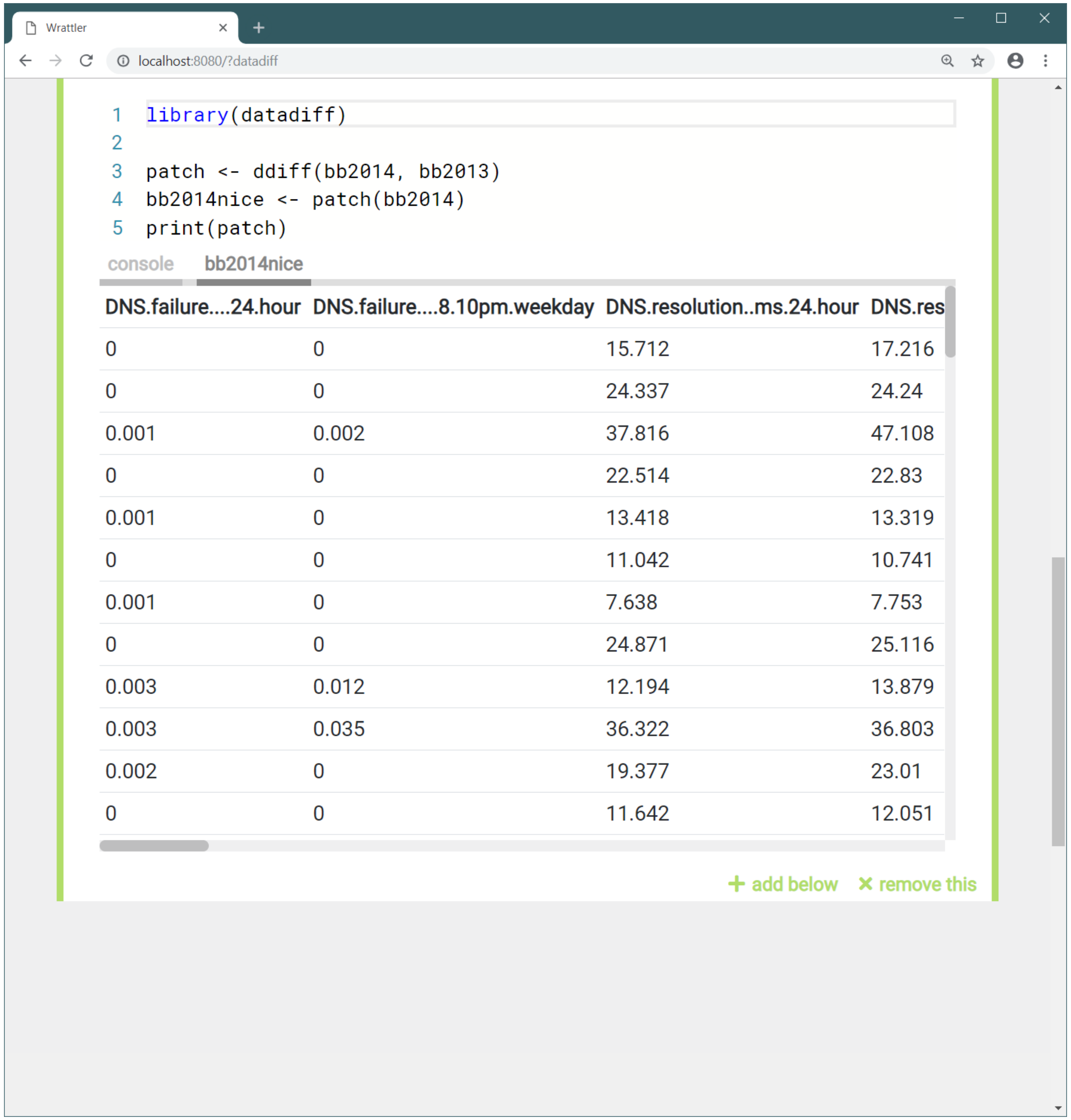


```
library(datadiff)

patch <- ddiff(bb2014, bb2013)
bb2014nice <- patch(bb2014)
print(patch)
```

```
Composed patch with elementary constituents:
Rescale patch.
  cols: 21
  shift: -0.349
  scale_factor: 446.788
Rescale patch.
  cols: 22
  shift: 0.337
  scale_factor: 123.932
Recode patch.
  cols: 23
  encoding: ADSL -> DSL, Cable -> ADSL2+, FTTC -> FTTx, FTTP ->
  one_to_one: TRUE
Delete patch.
  cols: 17
Delete patch.
  cols: 6
Permute patch.
  10 -> 11
  11 -> 10
  12 -> 16
  16 -> 18
  18 -> 19
  19 -> 22
  21 -> 23
  22 -> 25
  23 -> 26
  25 -> 27
```

Datadiff takes two datasets and generates **a list of patches** that can be applied to transform the structure of the first dataset into the structure of the second dataset.

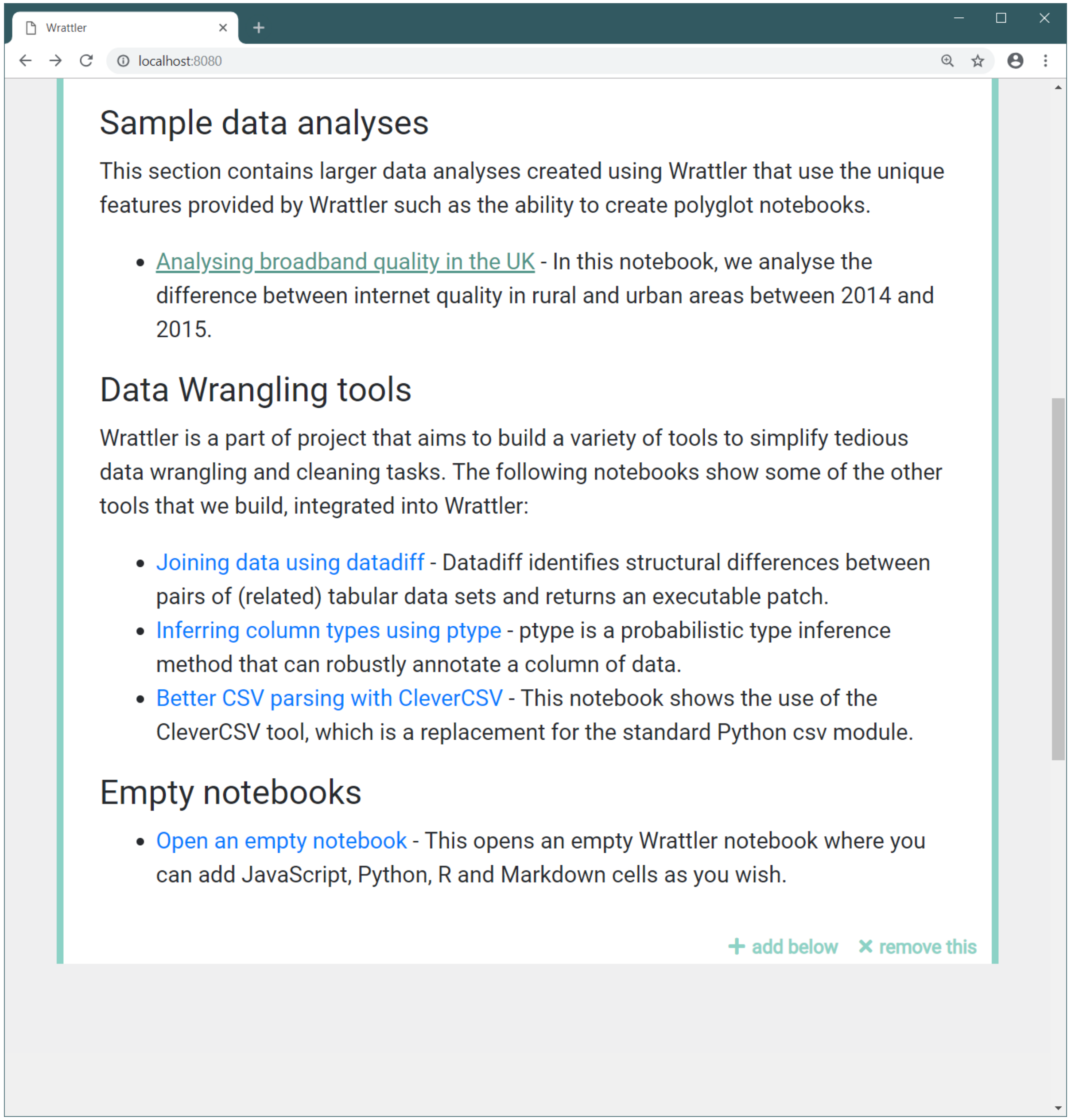
We first print the inferred patches.



```
library(datadiff)

patch <- ddiff(bb2014, bb2013)
bb2014nice <- patch(bb2014)
print(patch)
```

DNS.failure....24.hour	DNS.failure....8.10pm.weekday	DNS.resolution..ms.24.hour	DNS.res
0	0	15.712	17.216
0	0	24.337	24.24
0.001	0.002	37.816	47.108
0	0	22.514	22.83
0.001	0	13.418	13.319
0	0	11.042	10.741
0.001	0	7.638	7.753
0	0	24.871	25.116
0.003	0.012	12.194	13.879
0.003	0.035	36.322	36.803
0.002	0	19.377	23.01
0	0	11.642	12.051

We can switch the tab to **bb2014nice** to see the new transformed dataset, which is now compatible with the **bb2013** dataset.



Sample data analyses

This section contains larger data analyses created using Wrattler that use the unique features provided by Wrattler such as the ability to create polyglot notebooks.

- Analysing broadband quality in the UK - In this notebook, we analyse the difference between internet quality in rural and urban areas between 2014 and 2015.

Data Wrangling tools

Wrattler is a part of project that aims to build a variety of tools to simplify tedious data wrangling and cleaning tasks. The following notebooks show some of the other tools that we build, integrated into Wrattler:

- Joining data using datadiff - Datadiff identifies structural differences between pairs of (related) tabular data sets and returns an executable patch.
- Inferring column types using ptype - ptype is a probabilistic type inference method that can robustly annotate a column of data.
- Better CSV parsing with CleverCSV - This notebook shows the use of the CleverCSV tool, which is a replacement for the standard Python csv module.

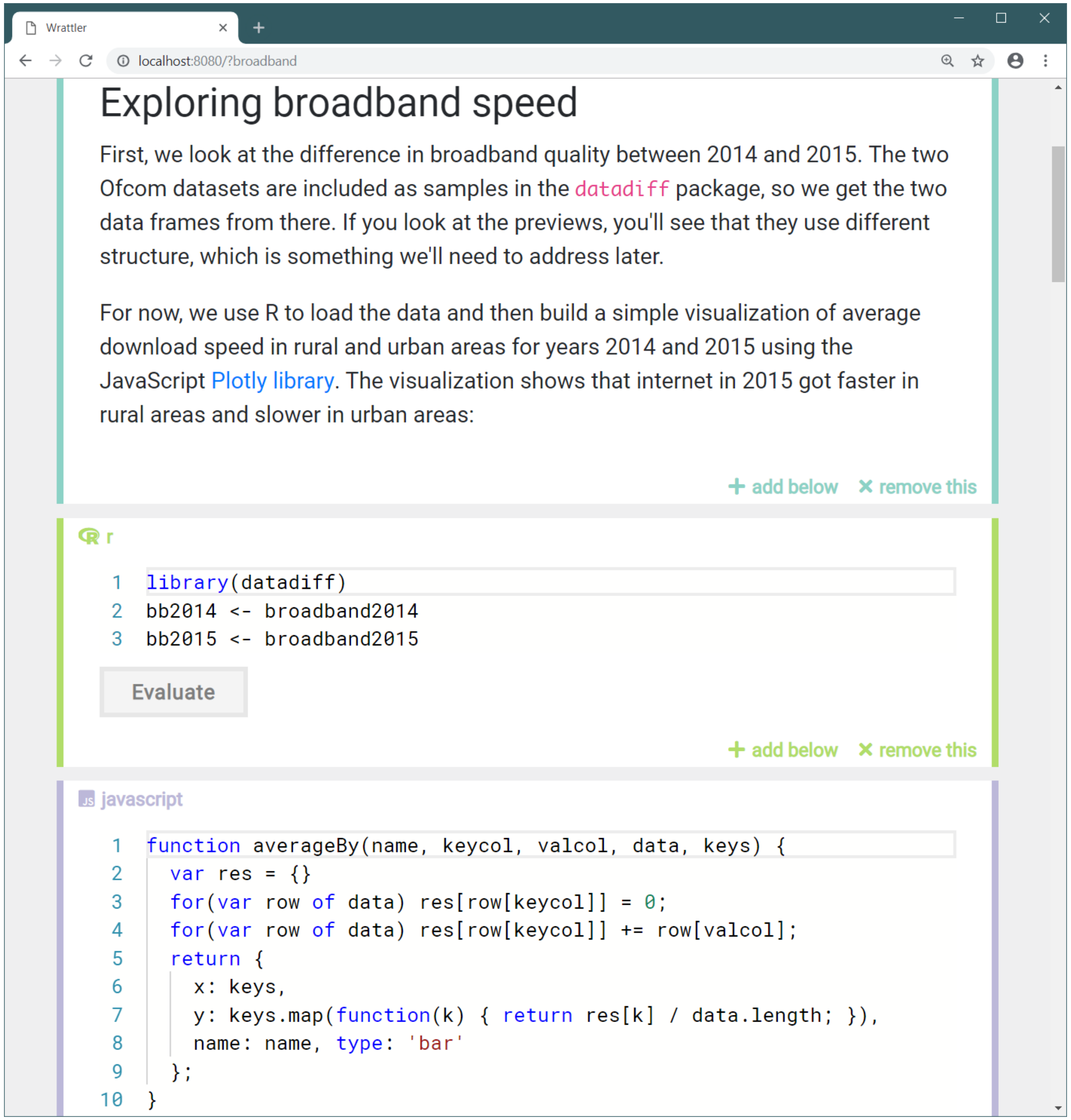
Empty notebooks

- Open an empty notebook - This opens an empty Wrattler notebook where you can add JavaScript, Python, R and Markdown cells as you wish.

In addition to **datadiff,** Wrattler also comes with examples showing **ptype** for inferring types of columns and **CleverCSV** for smart CSV parsing.

Comprehensive

Simplifying the data analytics process



Exploring broadband speed

First, we look at the difference in broadband quality between 2014 and 2015. The two Ofcom datasets are included as samples in the `datadiff` package, so we get the two data frames from there. If you look at the previews, you'll see that they use different structure, which is something we'll need to address later.

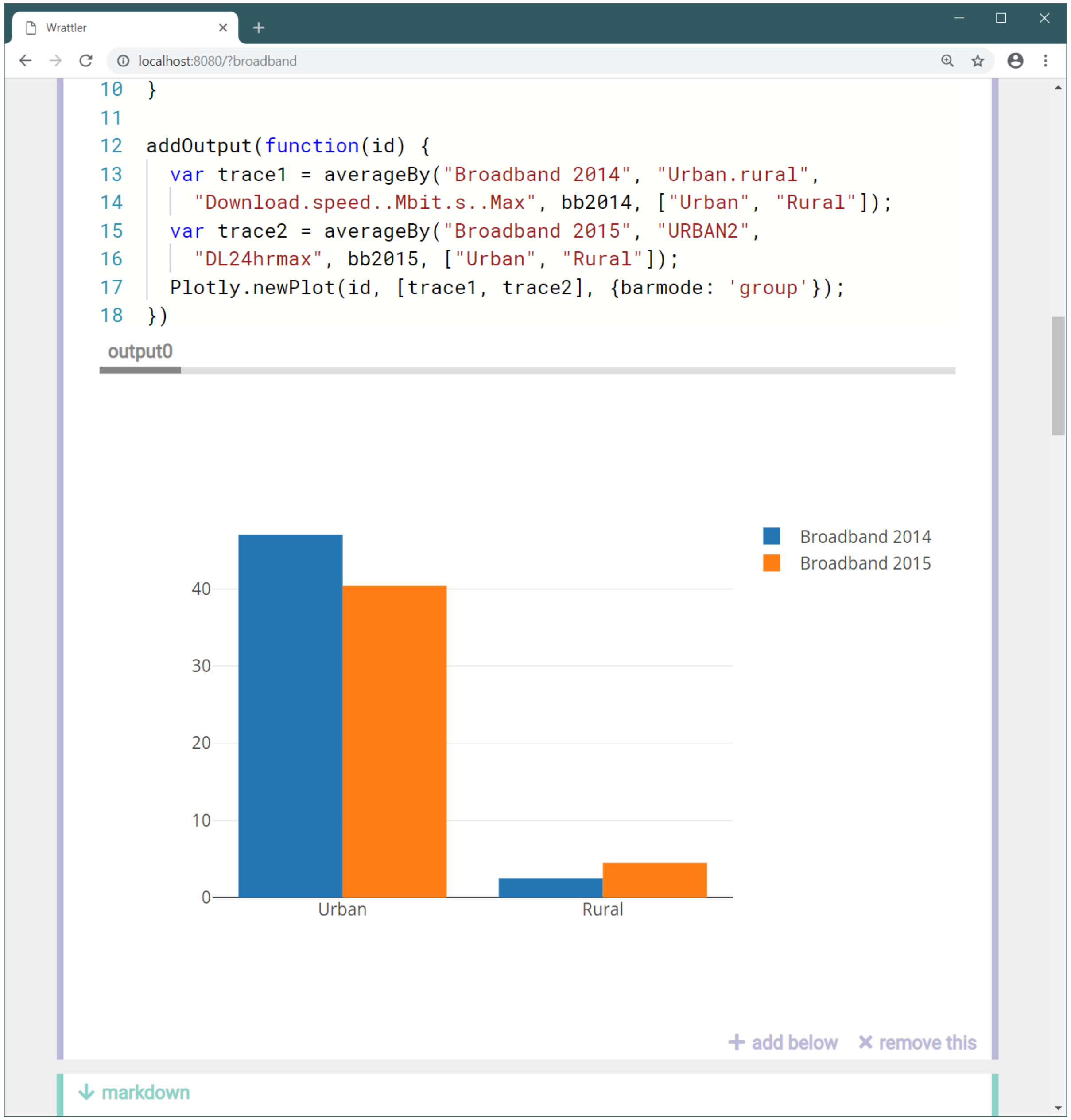
For now, we use R to load the data and then build a simple visualization of average download speed in rural and urban areas for years 2014 and 2015 using the JavaScript Plotly library. The visualization shows that internet in 2015 got faster in rural areas and slower in urban areas:

```r
library(datadiff)
bb2014 <- broadband2014
bb2015 <- broadband2015
```

```javascript
function averageBy(name, keycol, valcol, data, keys) {
  var res = {}
  for(var row of data) res[row[keycol]] = 0;
  for(var row of data) res[row[keycol]] += row[valcol];
  return {
    x: keys,
    y: keys.map(function(k) { return res[k] / data.length; }),
    name: name, type: 'bar'
  };
}
```

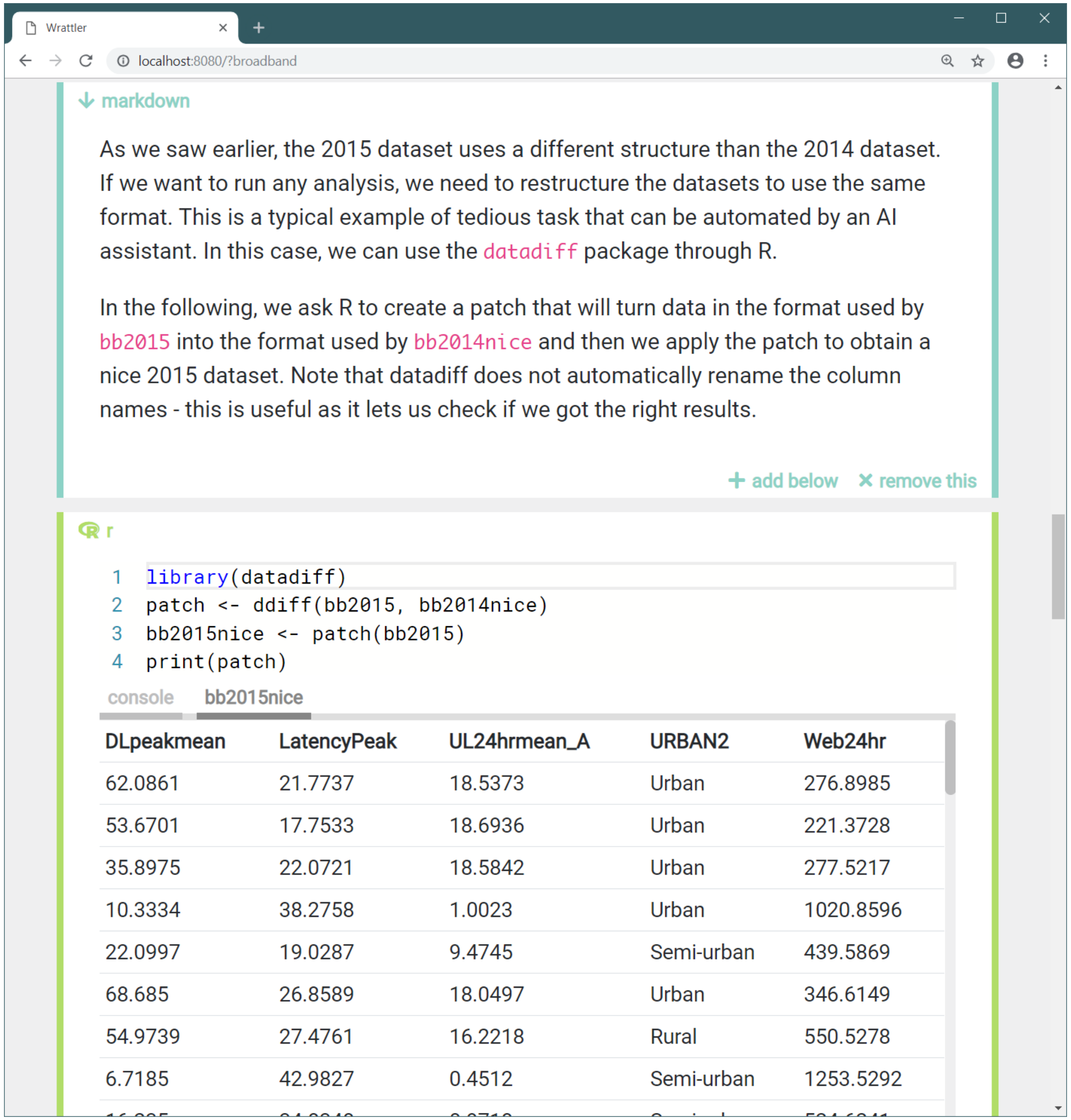
Wrattler helps with **all stages** of the data analytics process.

We look at a larger notebook, analysing the UK broadband quality data.



```
}

addOutput(function(id) {
  var trace1 = averageBy("Broadband 2014", "Urban.rural",
    "Download.speed..Mbit.s..Max", bb2014, ["Urban", "Rural"]);
  var trace2 = averageBy("Broadband 2015", "URBAN2",
    "DL24hrmax", bb2015, ["Urban", "Rural"]);
  Plotly.newPlot(id, [trace1, trace2], {barmode: 'group'});
})
```

We can mix **R and Python** for loading data with **JavaScript** for quickly visualizing and exploring data.



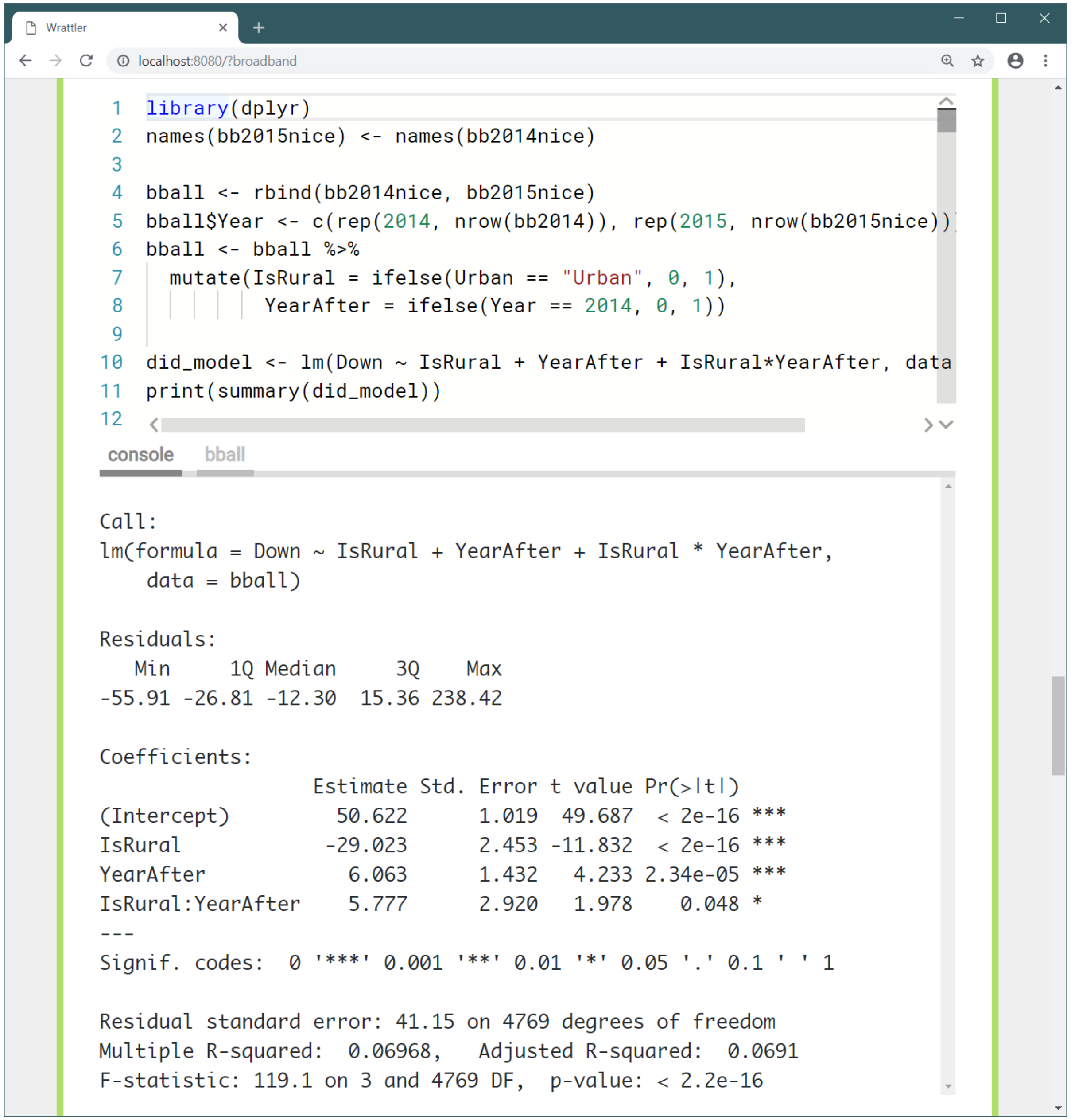
As we saw earlier, the 2015 dataset uses a different structure than the 2014 dataset. If we want to run any analysis, we need to restructure the datasets to use the same format. This is a typical example of tedious task that can be automated by an AI assistant. In this case, we can use the `datadiff` package through R.

In the following, we ask R to create a patch that will turn data in the format used by `bb2015` into the format used by `bb2014nice` and then we apply the patch to obtain a nice 2015 dataset. Note that datadiff does not automatically rename the column names - this is useful as it lets us check if we got the right results.

```
library(datadiff)
patch <- ddiff(bb2015, bb2014nice)
bb2015nice <- patch(bb2015)
print(patch)
```

DLpeakmean	LatencyPeak	UL24hrmean_A	URBAN2	Web24hr
62.0861	21.7737	18.5373	Urban	276.8985
53.6701	17.7533	18.6936	Urban	221.3728
35.8975	22.0721	18.5842	Urban	277.5217
10.3334	38.2758	1.0023	Urban	1020.8596
22.0997	19.0287	9.4745	Semi-urban	439.5869
68.685	26.8589	18.0497	Urban	346.6149
54.9739	27.4761	16.2218	Rural	550.5278
6.7185	42.9827	0.4512	Semi-urban	1253.5292

We have access to **datadiff** and other data wrangling tools, which help us make our data ready for interesting analytical tasks.



```
library(dplyr)
names(bb2015nice) <- names(bb2014nice)

bball <- rbind(bb2014nice, bb2015nice)
bball$Year <- c(rep(2014, nrow(bb2014)), rep(2015, nrow(bb2015nice))
bball <- bball %>%
  mutate(IsRural = ifelse(Urban == "Urban", 0, 1),
         YearAfter = ifelse(Year == 2014, 0, 1))

did_model <- lm(Down ~ IsRural + YearAfter + IsRural*YearAfter, data
print(summary(did_model))

```

```
Call:
lm(formula = Down ~ IsRural + YearAfter + IsRural * YearAfter,
    data = bball)

Residuals:
   Min      1Q Median      3Q     Max
-55.91 -26.81 -12.30  15.36 238.42

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)         50.622      1.019  49.687  < 2e-16 ***
IsRural            -29.023      2.453 -11.832  < 2e-16 ***
YearAfter            6.063      1.432   4.233 2.34e-05 ***
IsRural:YearAfter    5.777      2.920   1.978    0.048 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 41.15 on 4769 degrees of freedom
Multiple R-squared:  0.06968,	Adjusted R-squared:  0.0691
F-statistic: 119.1 on 3 and 4769 DF,  p-value: < 2.2e-16
```

Wrattler makes it easy to analyse data using the **wide range of libraries** available for R and Python.