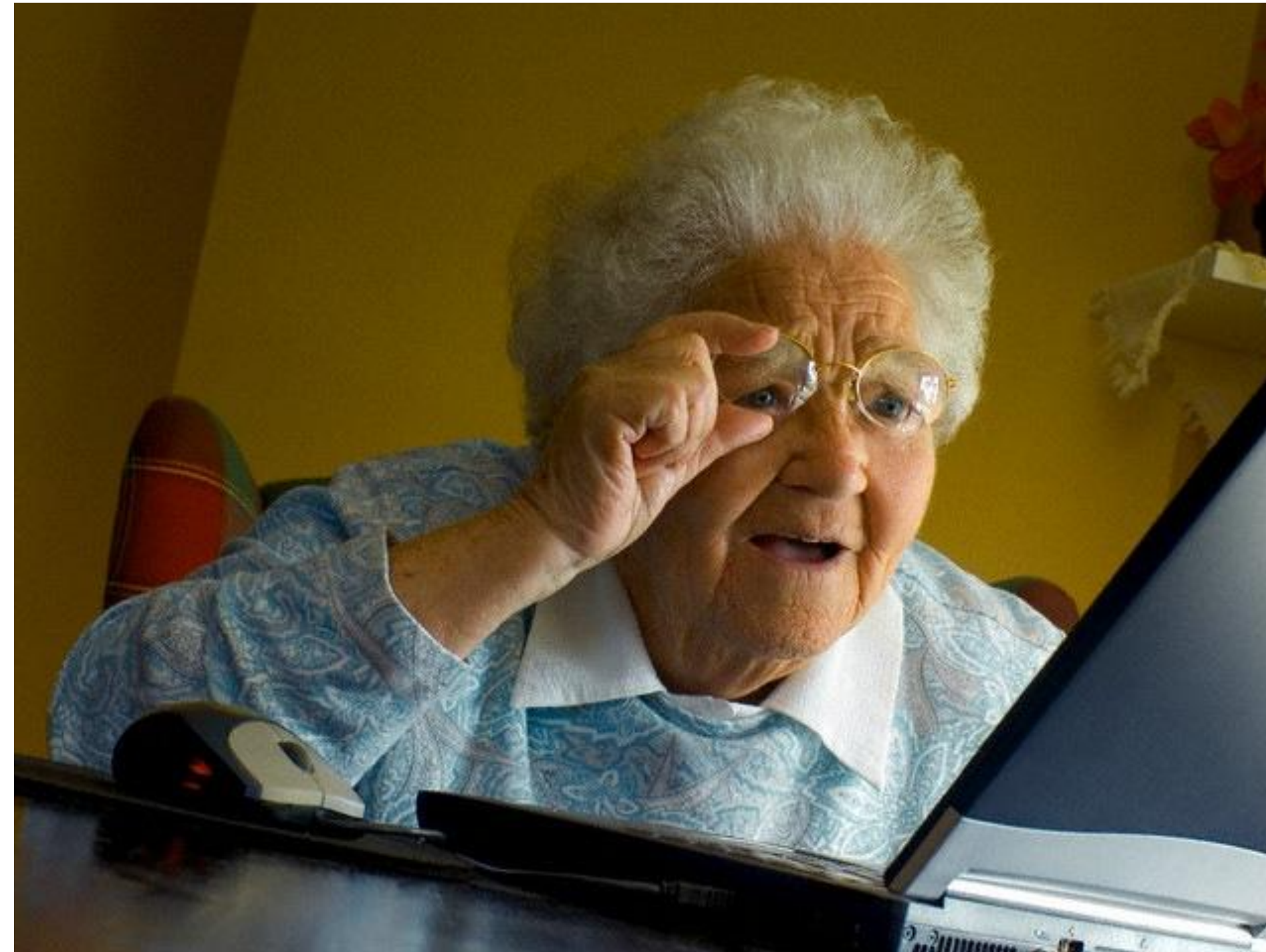


Studien bauen in lab.js

Abbildung 1
Es ist nie zu spät zu lernen, wie man 'ne richtig knorke Studie baut.



Imgflip, n.D.

Springschool März 2022 - Anfängertrack - Tag 1

Ablauf

Abbildung 2
Merle (rechts), wenn sie 3h nur über R reden darf.



lStock, 2017

1. Was ist der Unterschied zwischen lab.js und PsychoPy?
2. Basics in lab.js (inkl. kleinen Aufgaben für euch)
3. Ihr sagt, was ich bauen soll, und ich bau es. (Es wird super.)

lab.js vs PsychoPy

	PsychoPy	lab.js
Programmiersprache im Builder	Python	JavaScript
Muss man ein Programm installieren?	ja	nein (nur Google Chrome)
Muss man viel selbst programmieren?	eher ja	nein

Erste Schritte in lab.js

Builder:
Seite, auf der man die
Studien baut

Documentation:
Handbuch mit allen
Funktionen und Tutorials

Support:
Link zum Slack-Channel

lab.js Overview

Builder Documentation Support Resources ▾

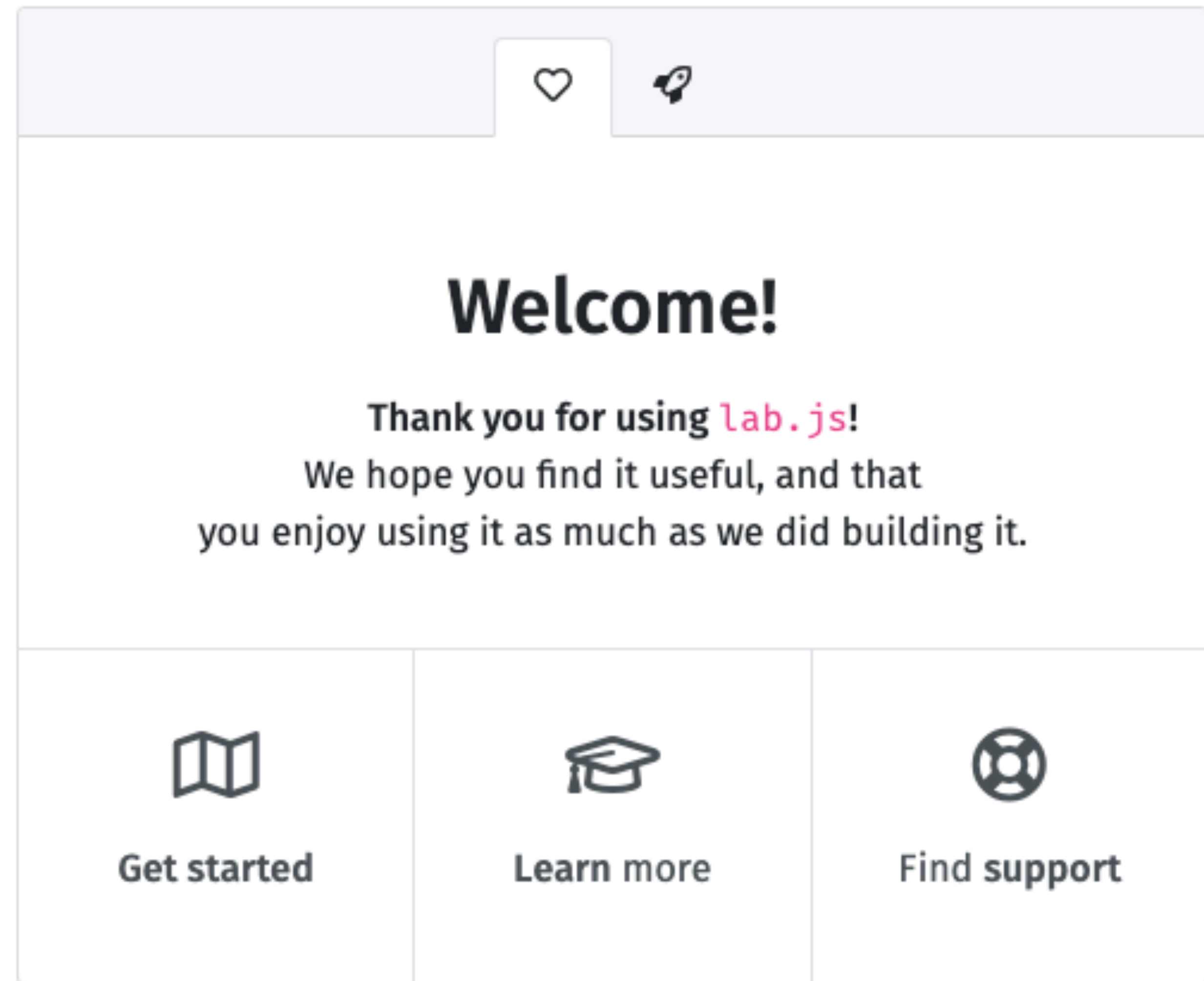
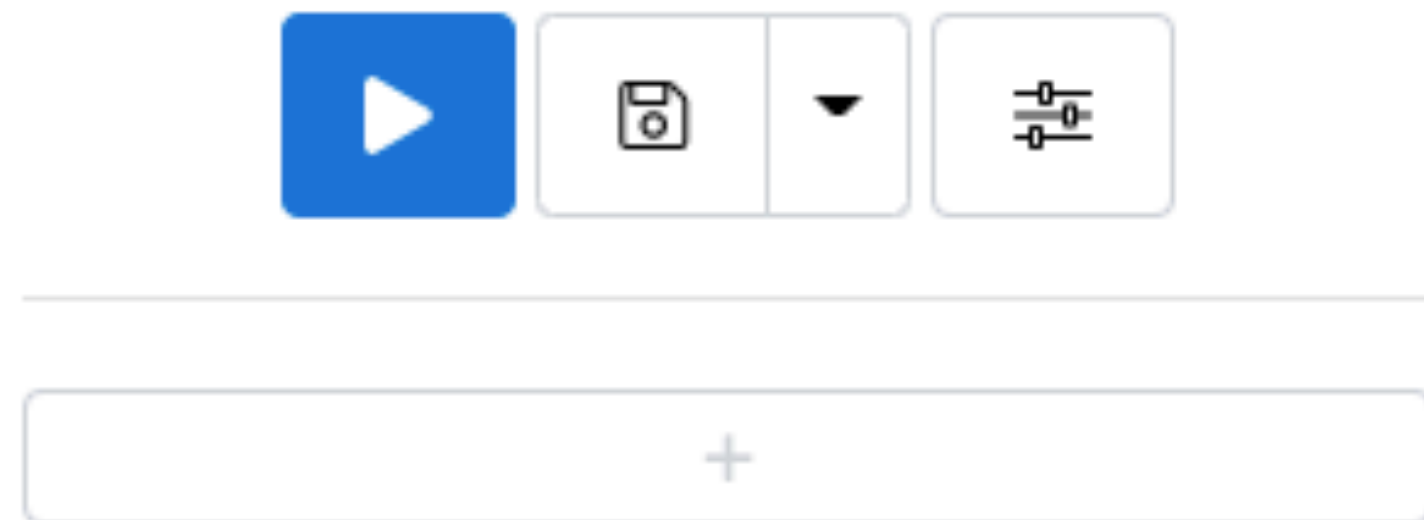


Online research made easy

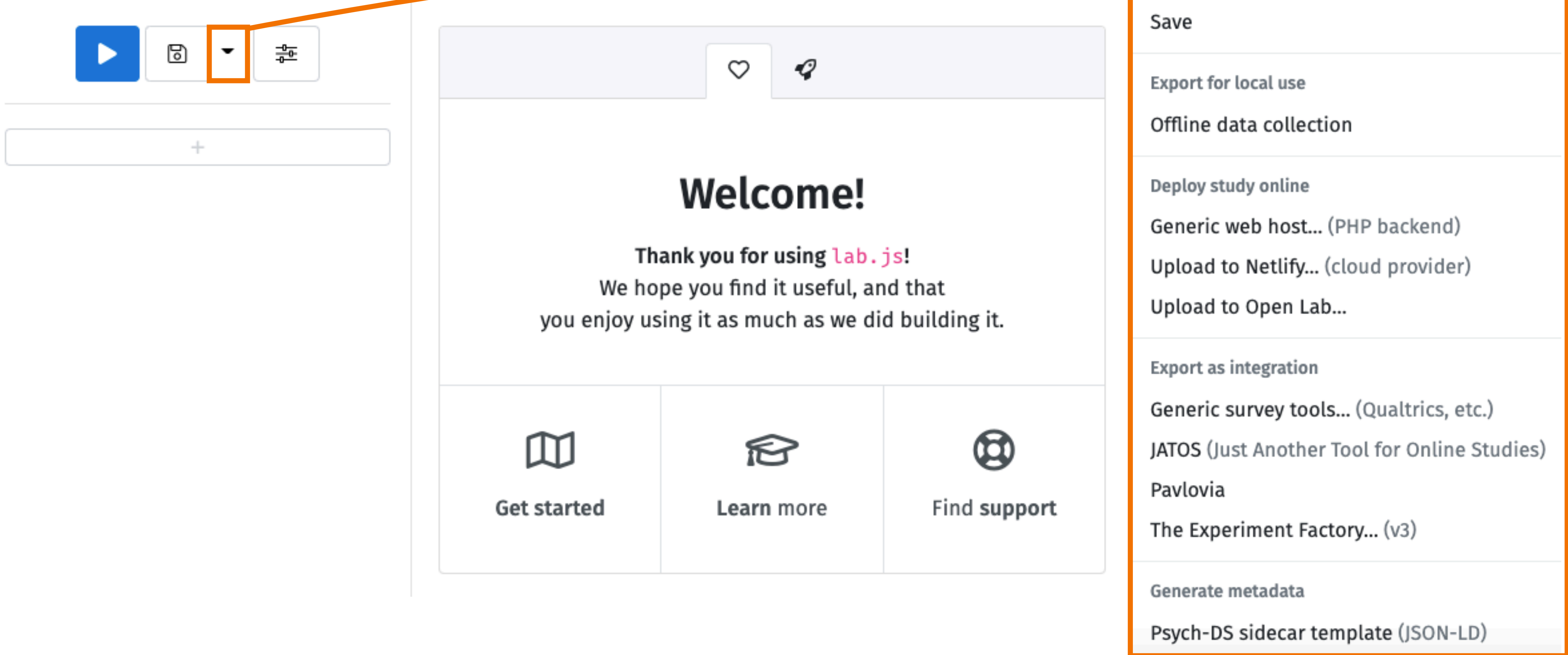
lab.js is a free, open, online study
builder for the behavioral and cognitive
sciences. (it works great in the lab, too)



Erste Schritte in lab.js



Erste Schritte in lab.js



The image shows the lab.js web application interface. On the left, a toolbar contains icons for play, save, a dropdown menu (highlighted with an orange box), and settings. Below the toolbar is a search bar with a plus icon. The main content area displays a welcome message and three action buttons: 'Get started', 'Learn more', and 'Find support'. On the right, a dropdown menu is open, showing various options for study management and deployment. An orange arrow points from the dropdown icon in the toolbar to the open menu.

Study

- New
- Open
- Save

Export for local use

- Offline data collection

Deploy study online

- Generic web host... (PHP backend)
- Upload to Netlify... (cloud provider)
- Upload to Open Lab...

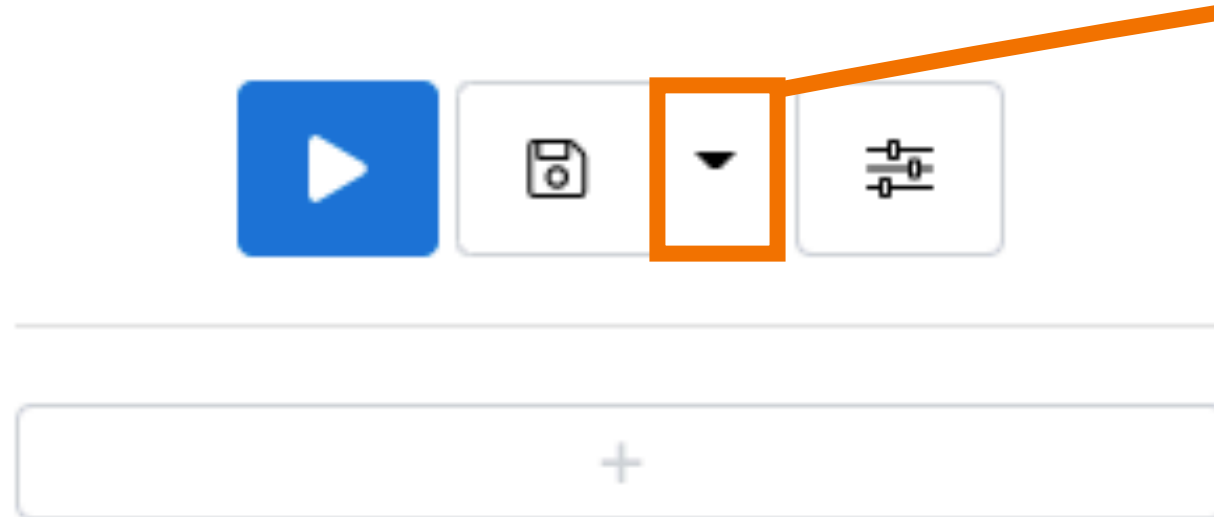
Export as integration

- Generic survey tools... (Qualtrics, etc.)
- JATOS (Just Another Tool for Online Studies)
- Pavlovia
- The Experiment Factory... (v3)

Generate metadata

- Psych-DS sidecar template (JSON-LD)

Erste Schritte in lab.js



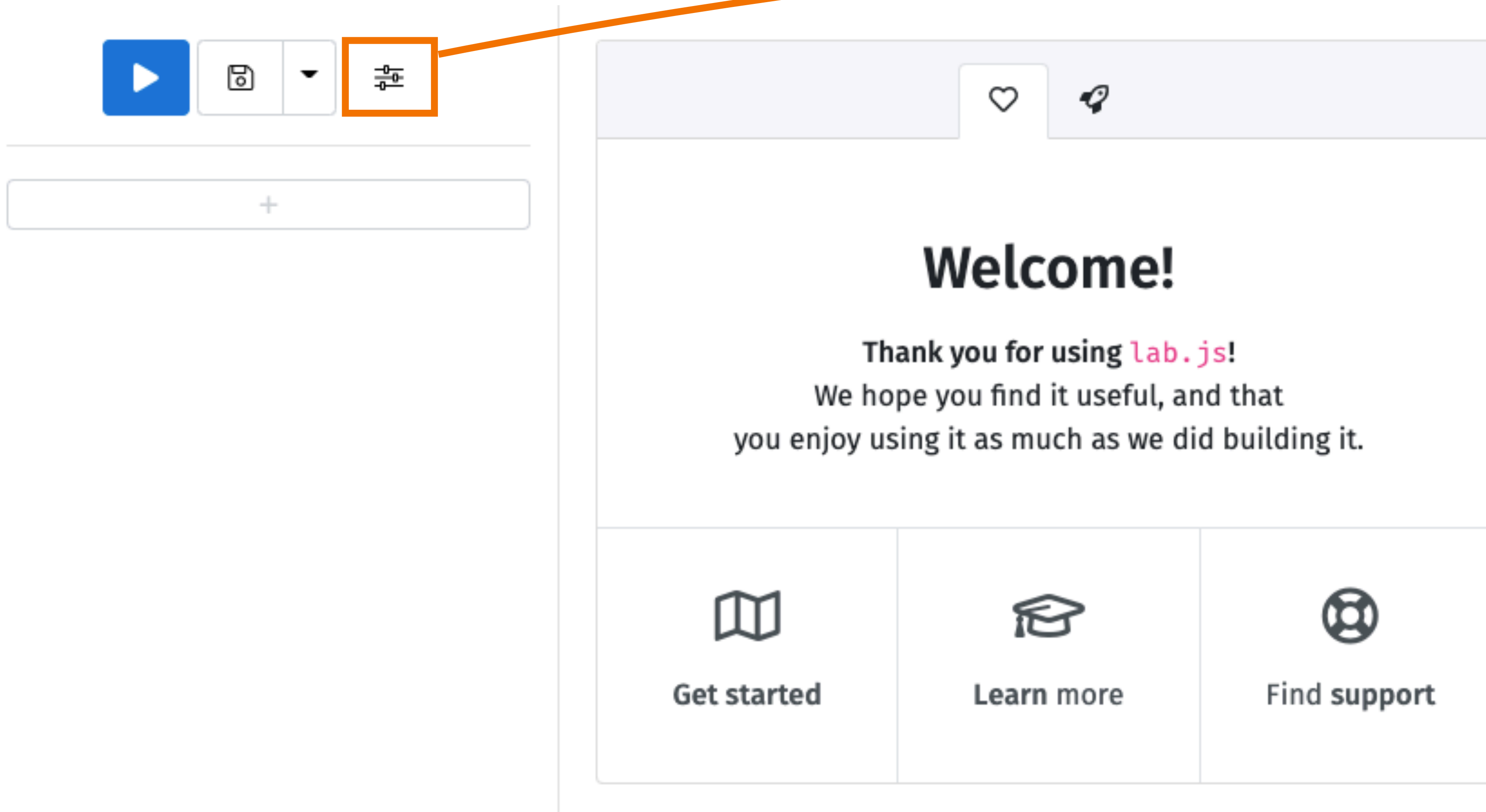
Studie als json-File
speichern

Studie als Laborstudie
exportieren

Studie als Onlinestudie
exportieren

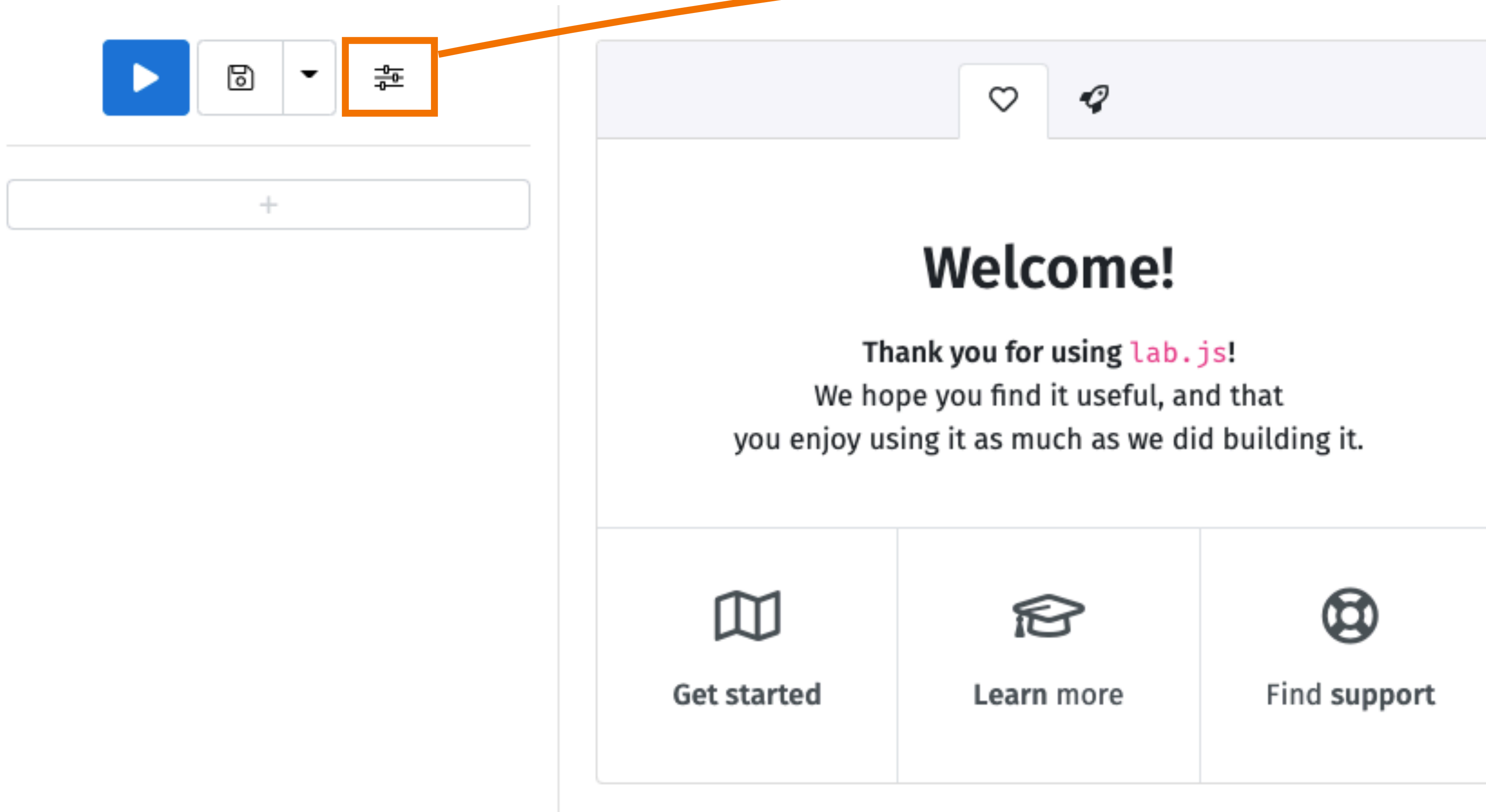
- Study
 - New
 - Open
 - Save
 - Export for local use
 - Offline data collection
 - Deploy study online
 - Generic web host... (PHP backend)
 - Upload to Netlify... (cloud provider)
 - Upload to Open Lab...
 - Export as integration
 - Generic survey tools... (Qualtrics, etc.)
 - JATOS (Just Another Tool for Online Studies)
 - Pavlovia
 - The Experiment Factory... (v3)
 - Generate metadata
 - Psych-DS sidecar template (JSON-LD)

Erste Schritte in lab.js



The screenshot shows a browser window with tabs for 'HTML' and 'CSS'. The page title is 'Study information'. Below the title is a prompt: 'Please tell your fellow scientists a few things about your study.' The form has three sections: 'Title' with a text input field, 'Description' with a large text area and the prompt 'What does your study do?', and 'Repository' with a text input field containing 'https://osf.io/...' and 'https://github.com/...'. Below the repository field is the prompt 'Where can researchers find the canonical or latest version of your study?'. The 'Contributors' section has a text input field containing 'Rita Levi-Montalcini <rlm@nobel.example>' and '(http://ibcn.cnr.it)'. The form is outlined with an orange border.

Erste Schritte in lab.js



i

HTML

CSS

✕

Storage in use

While storage is not strictly limited, we strongly recommend that studies use substantially less than 50 MB overall, and ideally below 20MB. This is not because of technical limitations but to reduce loading time and network traffic. If larger media are required, for example videos, these should be hosted outside of the study.

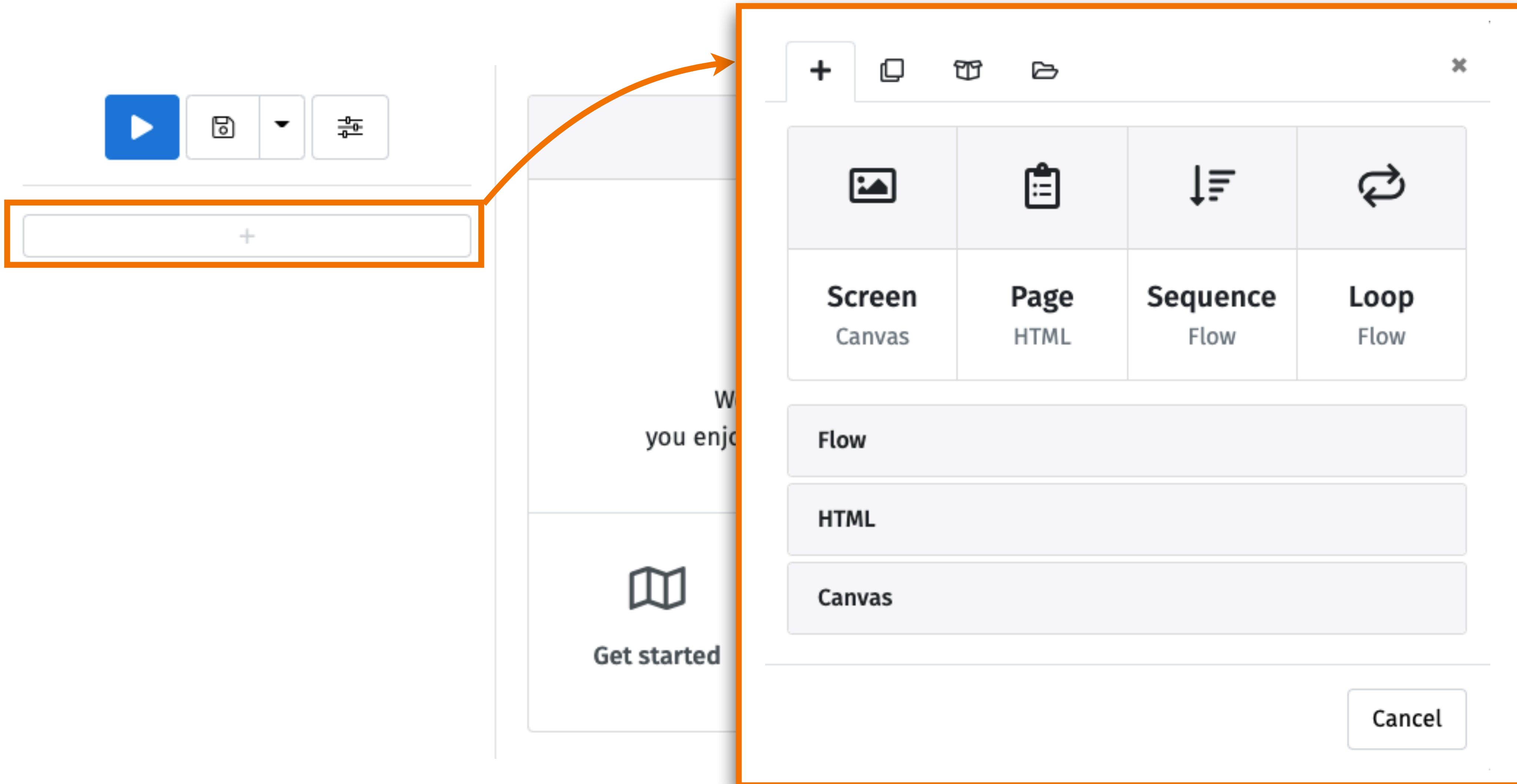
Study-wide static files

The following files are available study-wide from the **static** directory.

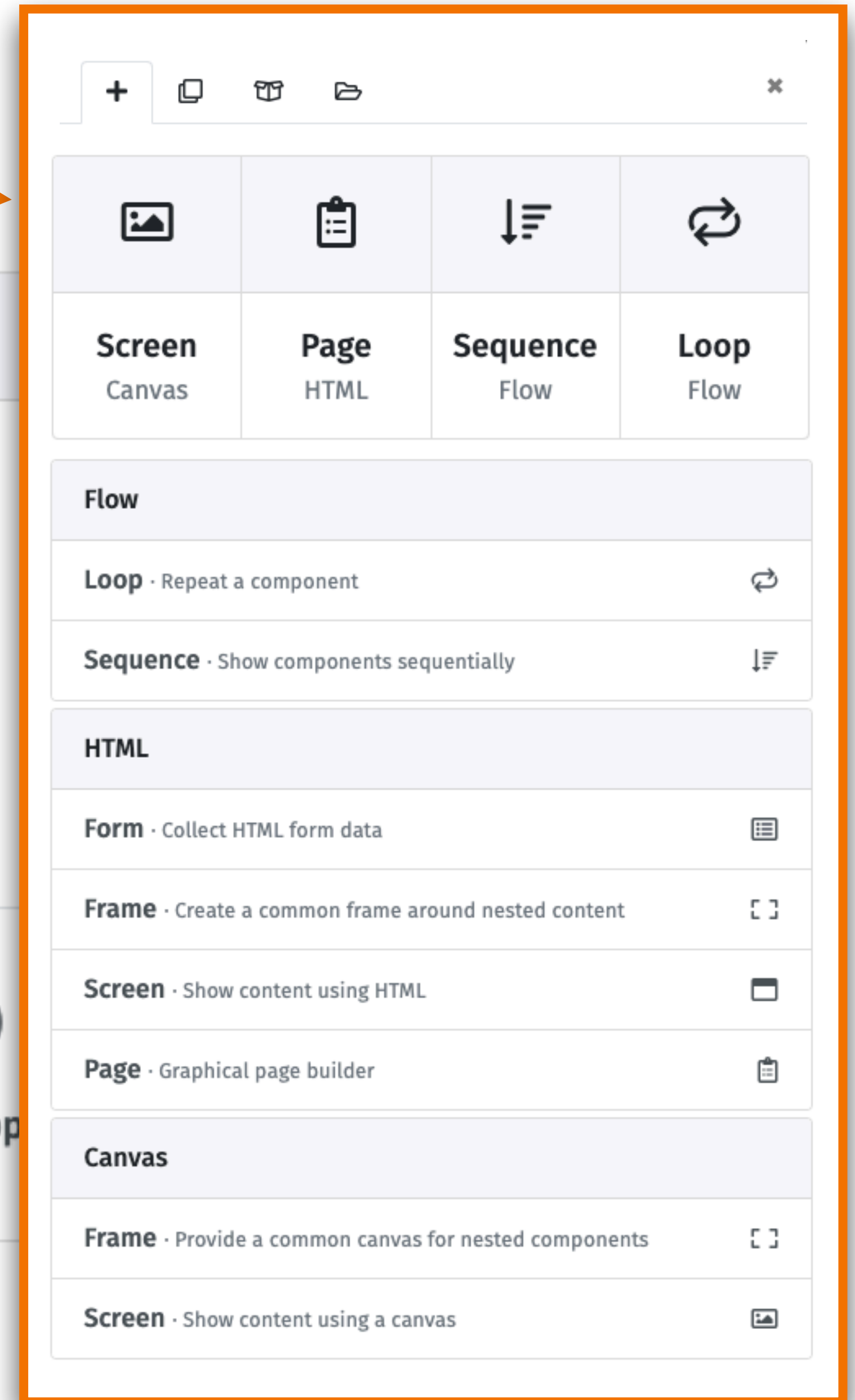
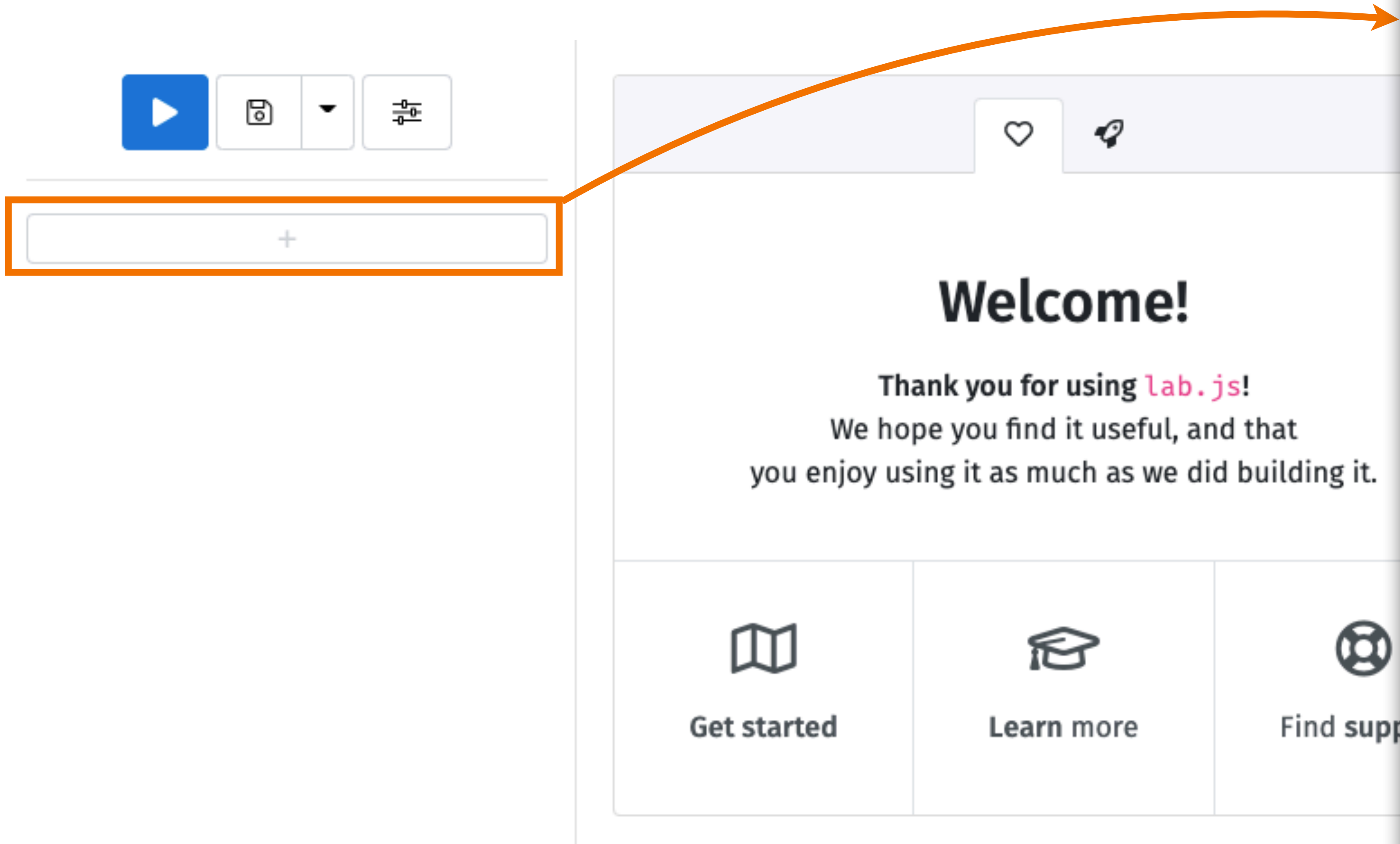
Filename	Size [KB]
+	

Done

Erste Schritte in lab.js



Erste Schritte in lab.js

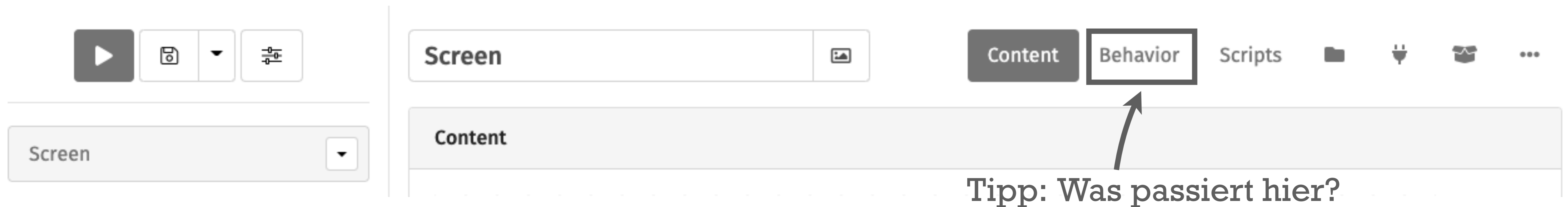


Erste Schritte in lab.js

Aufgabe 1:

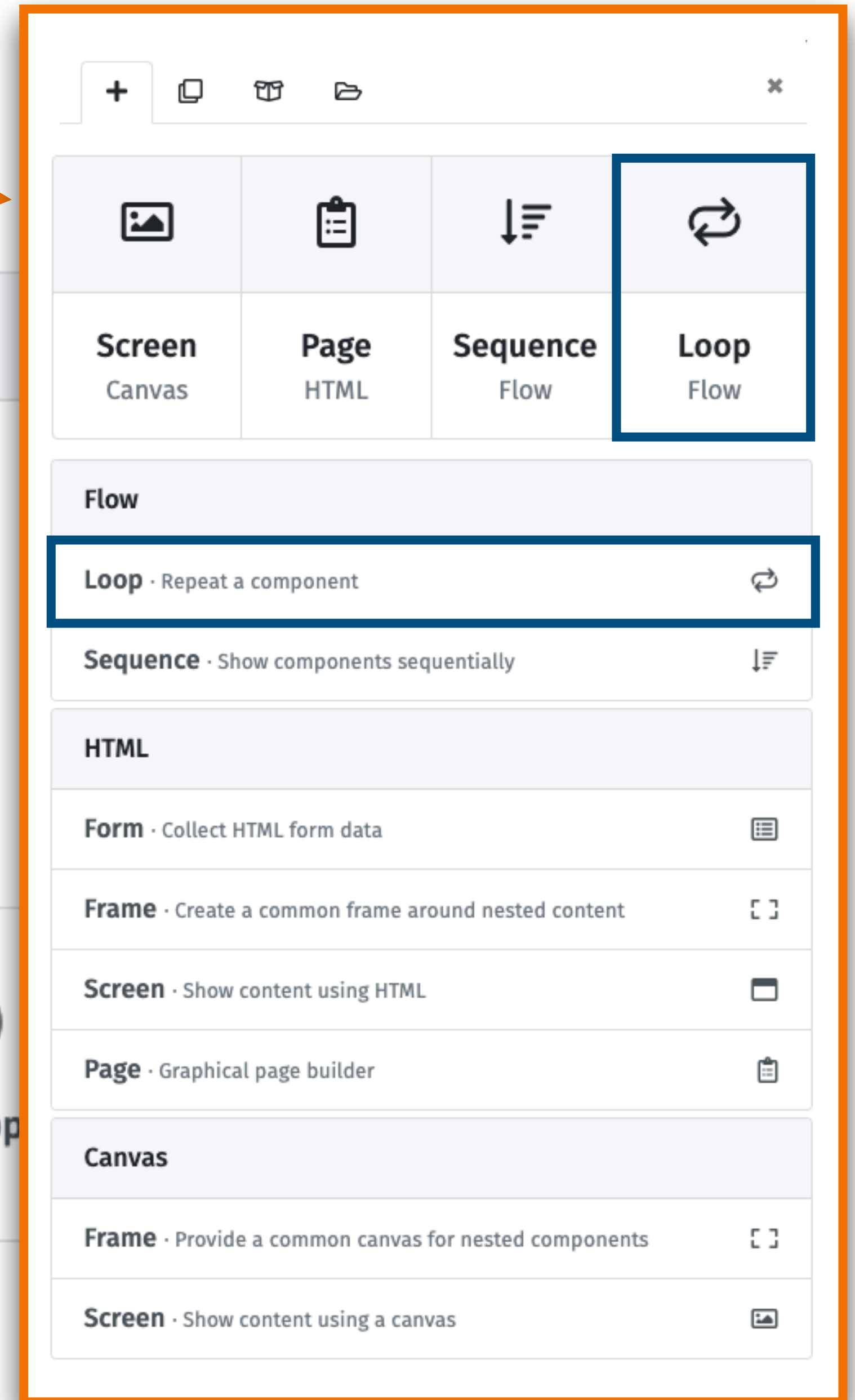
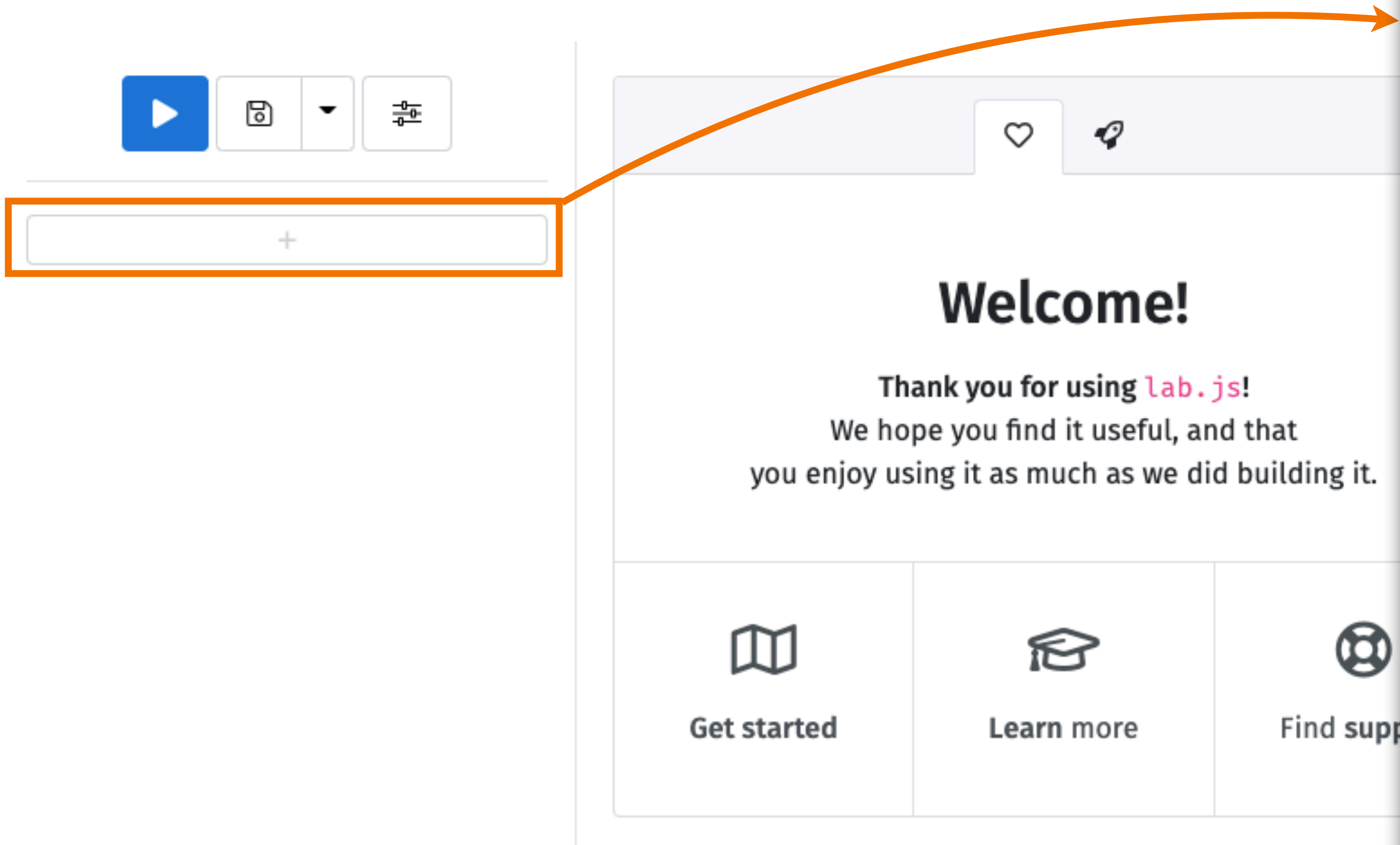
Baut eine Studie, in der zuerst eine multiple Choice Frage angezeigt wird und danach eine Seite mit einem roten Kreis in der Mitte.

Für Schnelle: Baut nach der Seite mit dem roten Kreis noch eine weitere in die Studie ein, auf der ein blauer Kreis angezeigt wird. Der rote Kreis sollte für 1 Sekunde angezeigt werden, bevor der blaue Kreis erscheint.



—> Wichtig: Merkt euch, wie ihr eure Studie gebaut habt und erzählt mir Schritt für Schritt, wie ich das Ganze nachbauen kann.

Erste Schritte in lab.js



Erste Schritte in lab.js

UVn +
ihre Faktorstufen (Bedingungen) +
ihr Datentyp



Loop

+

Hier kommt
alles rein,
was ihr
wiederholen wollt

Loop

Content

Behavior

Scripts

Loop

parameter0

A ▾

color

A ▾

+

≡

blue

🗑

≡

red

🗑

+

Sample

i

Use all

In random order

⬆

Wie viele
Wiederholungen
(= Trials) wollt ihr?

Wie sollen eure Trials abgespielt werden?
(z.B. random, Ziehen mit/ohne Zurücklegen, als
geordnete Sequenz)

Erste Schritte in lab.js

Aufgabe 2:

Baut eine Studie, in der zufällig ein roter oder ein blauer Kreis angezeigt werden (10 Trials).

Tipp 1: Nutzt einen Loop mit einer Variable „Farbname“

Tipp 2: Nutzt statt des Farbnamens `${parameters.Farbname}`

—> Wichtig: Merkt euch, wie ihr eure Studie gebaut habt und erzählt mir Schritt für Schritt, wie ich das Ganze nachbauen kann.

Erste Schritte in lab.js

Aufgabe 3:

Benutzt die gleiche Studie wie eben. Zwischen den Kreisen sollte aber diesmal jeweils für 1500 ms eine leere Seite angezeigt werden. Die Kreise sollten je nur für 30 ms angezeigt werden.

Tipp: In einen Loop passt nur eine Seite.

Wie könnt ihr trotzdem 2 Seiten unterbringen? Sucht nach einer zusätzlichen Komponente, die das Problem löst!

—> Wichtig: Merkt euch, wie ihr eure Studie gebaut habt und erzählt mir Schritt für Schritt, wie ich das Ganze nachbauen kann.

Erste Schritte in lab.js

Aufgabe 4:

Benutzt die gleiche Studie wie eben. Wir möchten jetzt, dass unsere VP auf die Leertaste drückt, wenn ein roter Kreis angezeigt wurde und nicht reagiert, wenn ein blauer Kreis angezeigt wurde. Wir möchten die Reaktion als „saw_red_circle“ speichern.

Tipp 1: Die VP sollte in der Pause zwischen den Kreisen reagieren.

Tipp 2: Schaut im Tab „Behavior“ nach.

—> Wichtig: Merkt euch, wie ihr eure Studie gebaut habt und erzählt mir Schritt für Schritt, wie ich das Ganze nachbauen kann.

Wir bauen eine Studie zum RSE

Aufgabe 5: Wir bauen eine kleine Studie zusammen! (Ich baue, ihr sagt mir, was wir brauchen!)

Experiment: Redundant Signals Effect

Es wird random entweder ein weißer Kreis angezeigt oder kein Kreis. Zusätzlich wird ein Klickgeräusch abgespielt oder kein Geräusch. Es gibt also 3 Bedingungen: V, A und VA

Die VP soll reagieren, wenn sie ein Klicken gehört, einen Kreis gesehen hat oder beides gleichzeitig wahrgenommen hat.

Datenanalyse in R

Abbildung 3
Niemand mag Excel (oder SPSS, wo wir schon dabei sind).



Imgflip, 2019

Springschool März 2022 - Anfängertrack - Tag 1

Ablauf

Abbildung 4
Merle (rechts), wenn sie 3h nur über R reden darf.

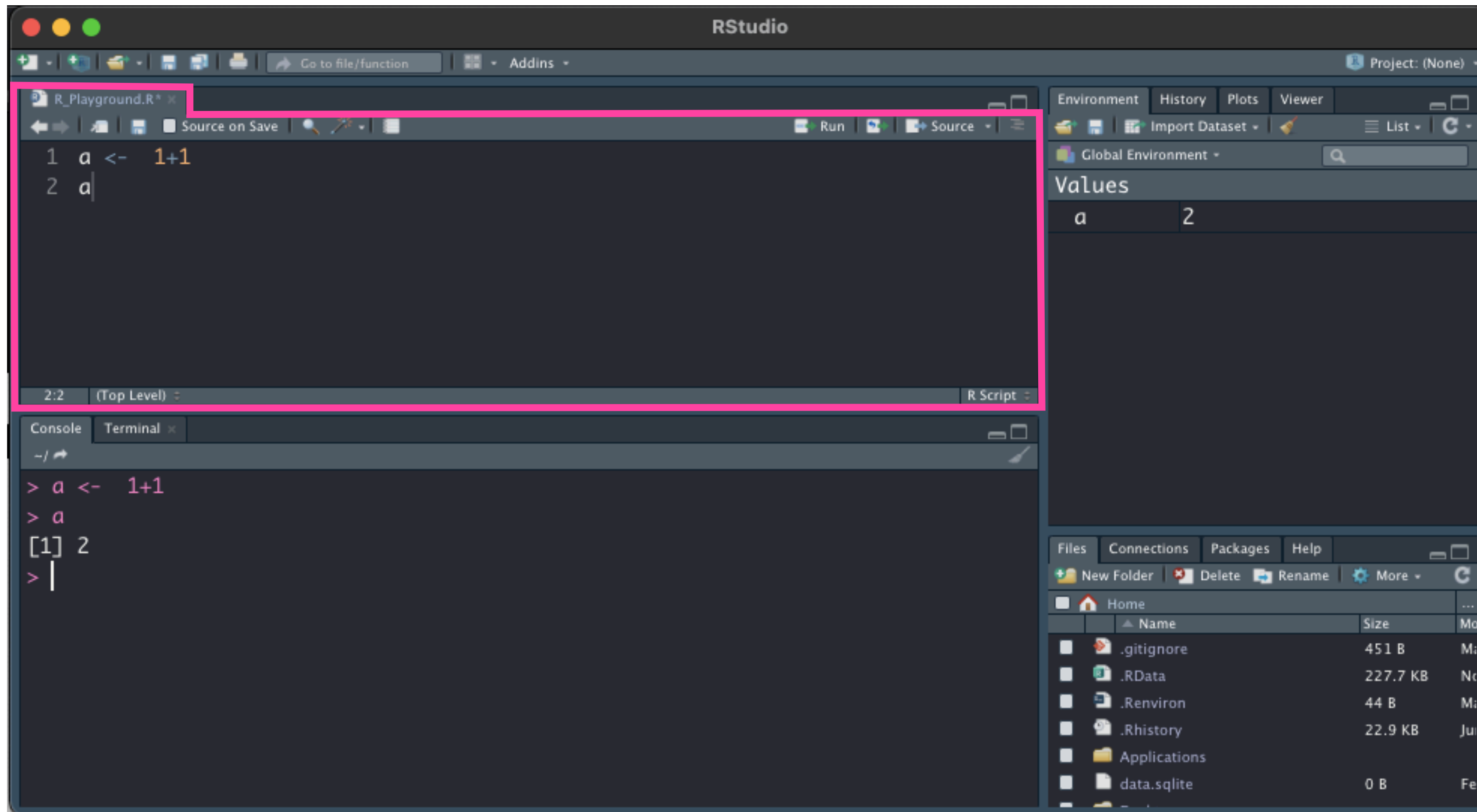


IStock, 2017

1. R-Studio
2. Basics
3. Funktionen
4. Loops
 - if-Loops
 - for-Loops
 - while-„Loops“
5. Ich programmiere ein Skript und erkläre euch Dinge. Es wird super.

R-Studio

Abbildung 3
R-Studio IDE



Skript:

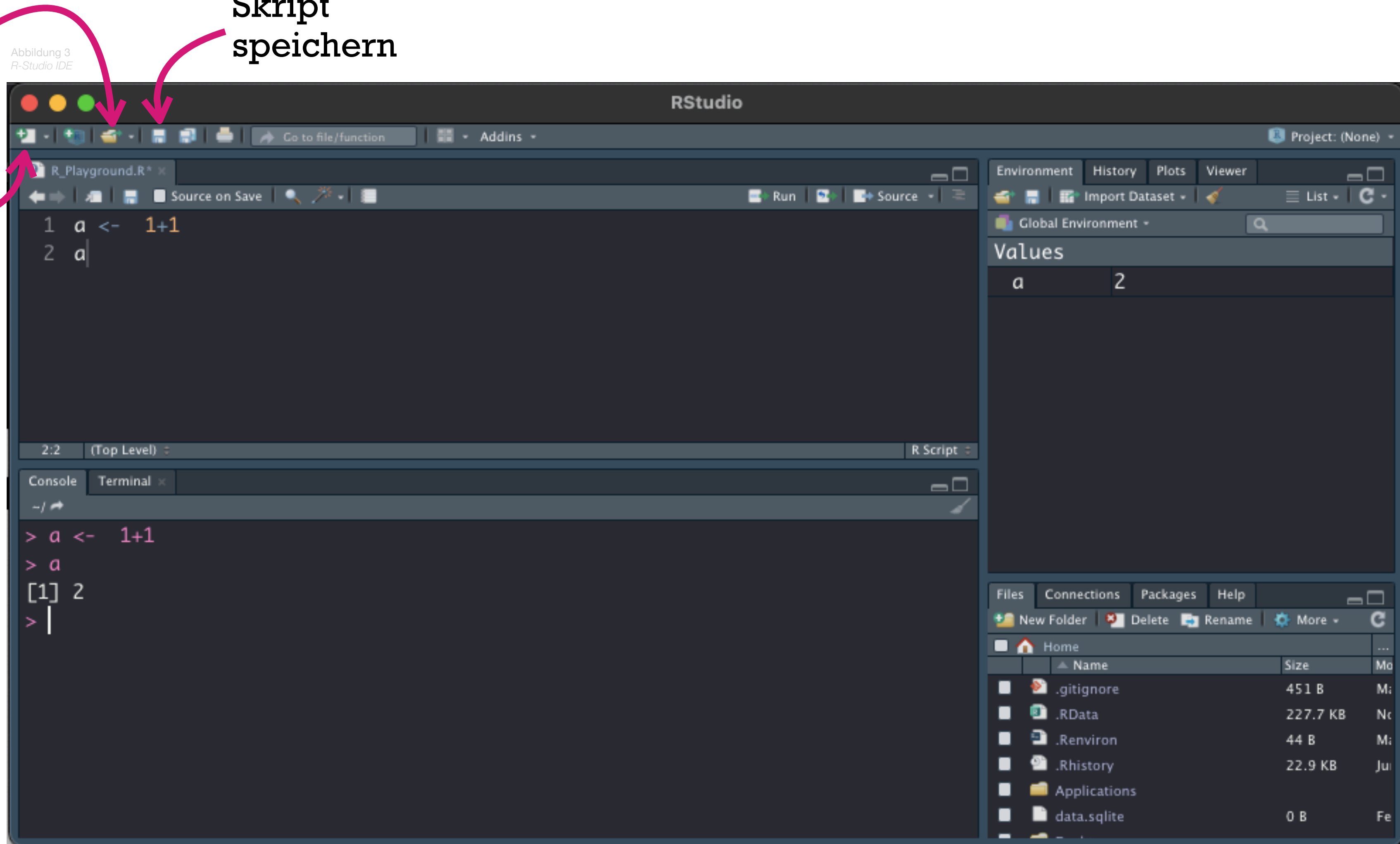
Das Dokument, in dem
euer fertiger Code steht

R-Studio

altes Skript
öffnen

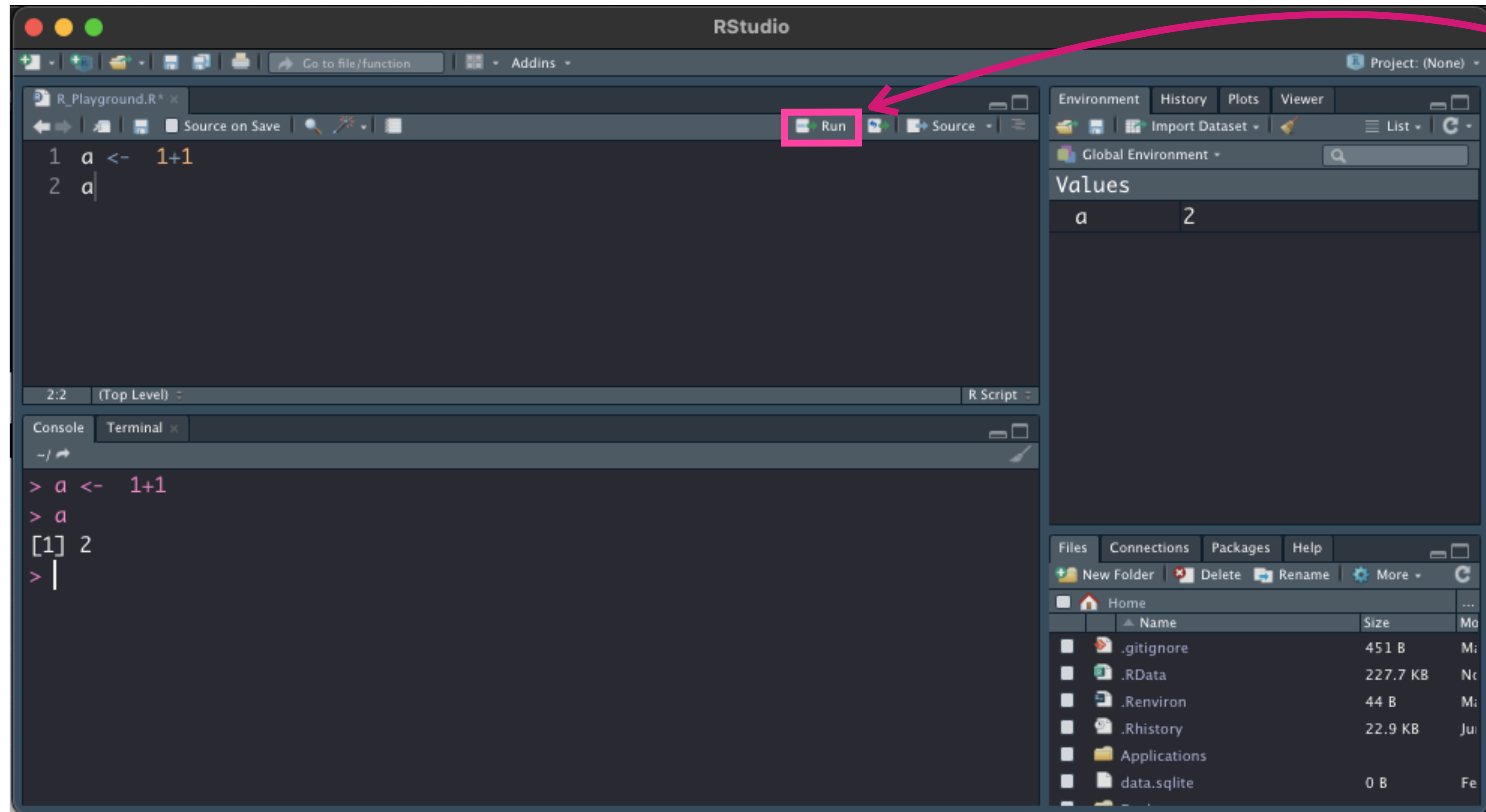
neues Skript
anlegen

Skript
speichern



R-Studio

Abbildung 3
R-Studio IDE

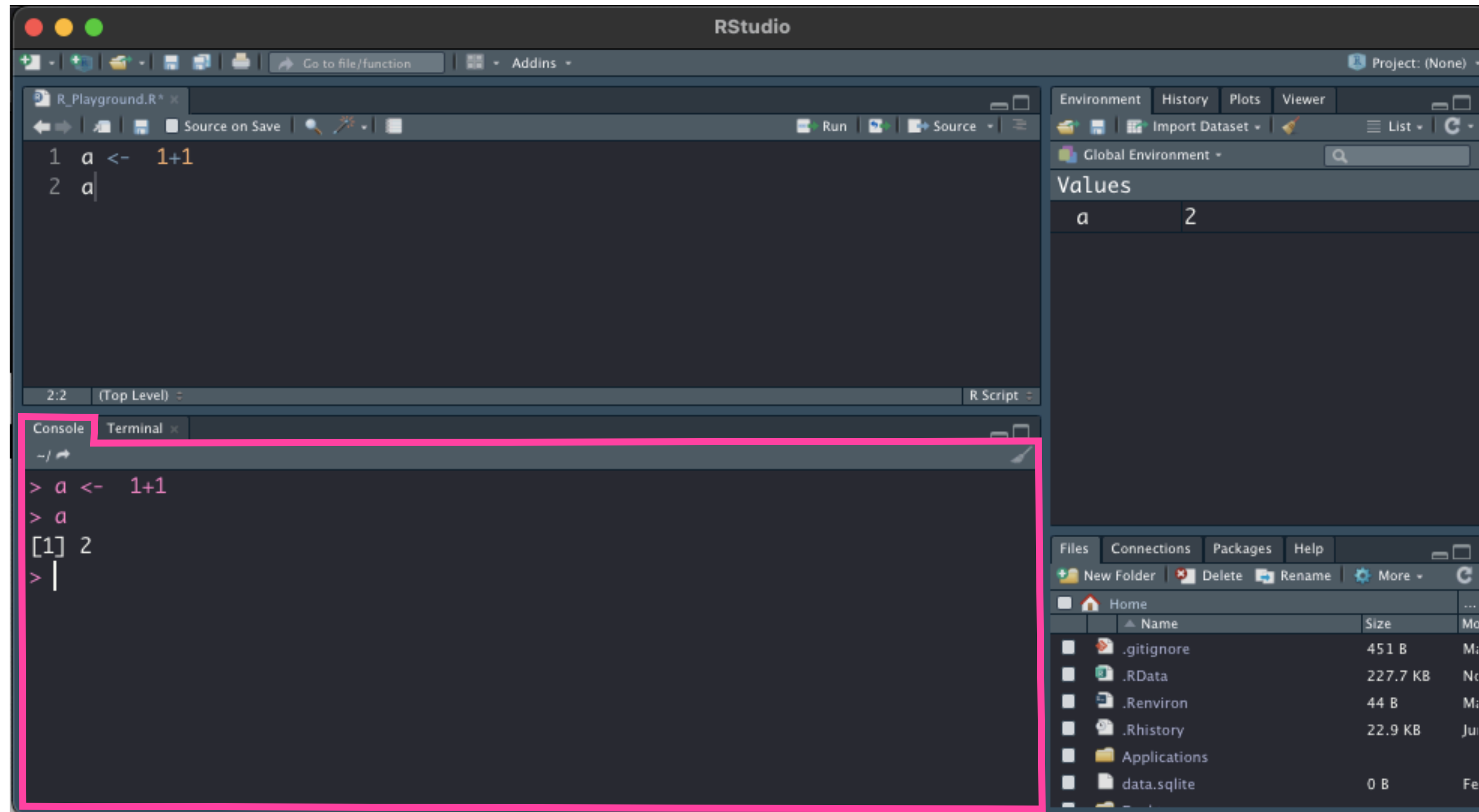


Run:
führt markierte
Zeile(n) im Skript
aus und gibt
Ergebnis in der
Konsole aus

Alternative:
alt + Enter

R-Studio

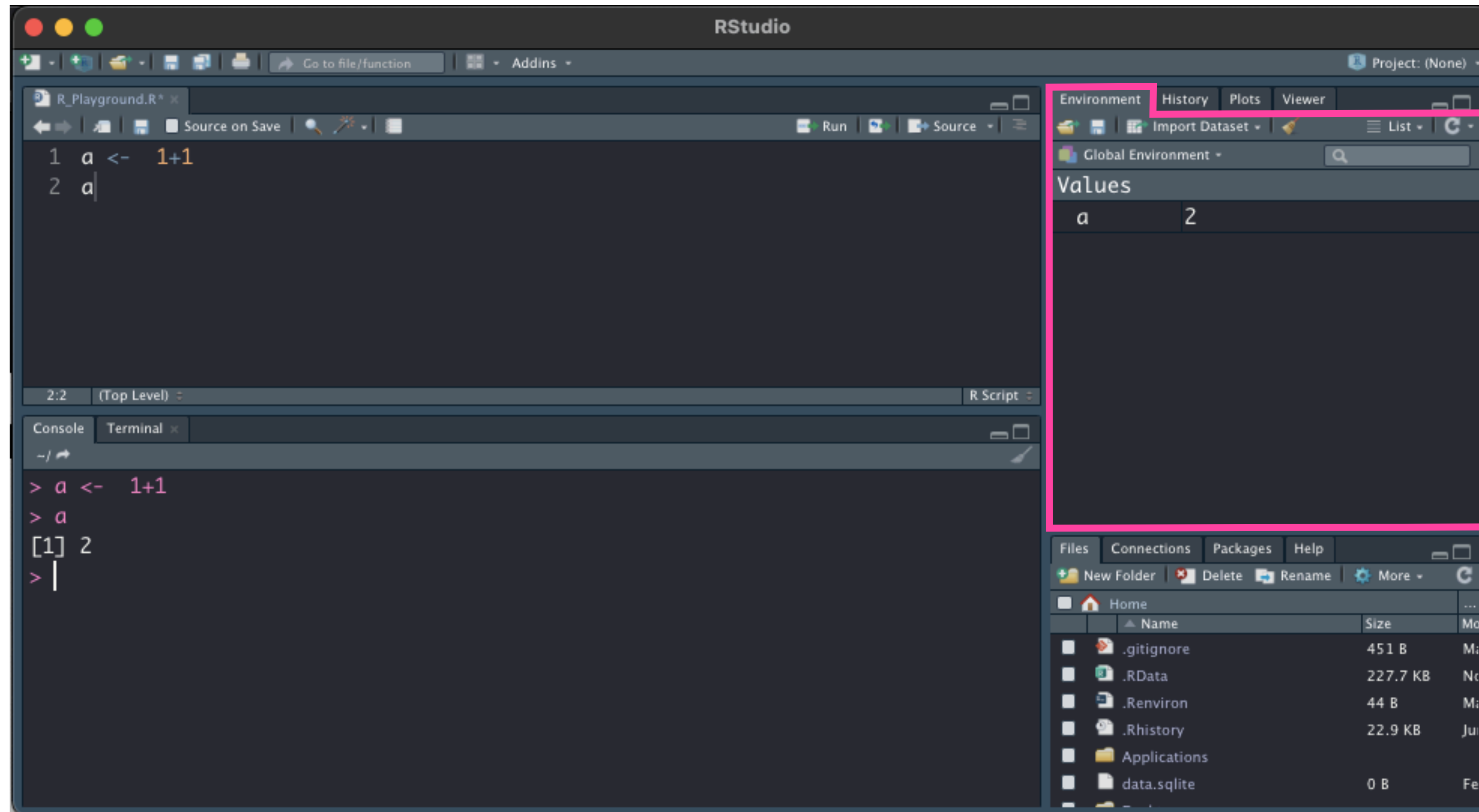
Abbildung 3
R-Studio IDE



Konsole:
gibt Ergebnisse und
Fehlermeldungen aus,
man kann auch darin
direkt schreiben

R-Studio

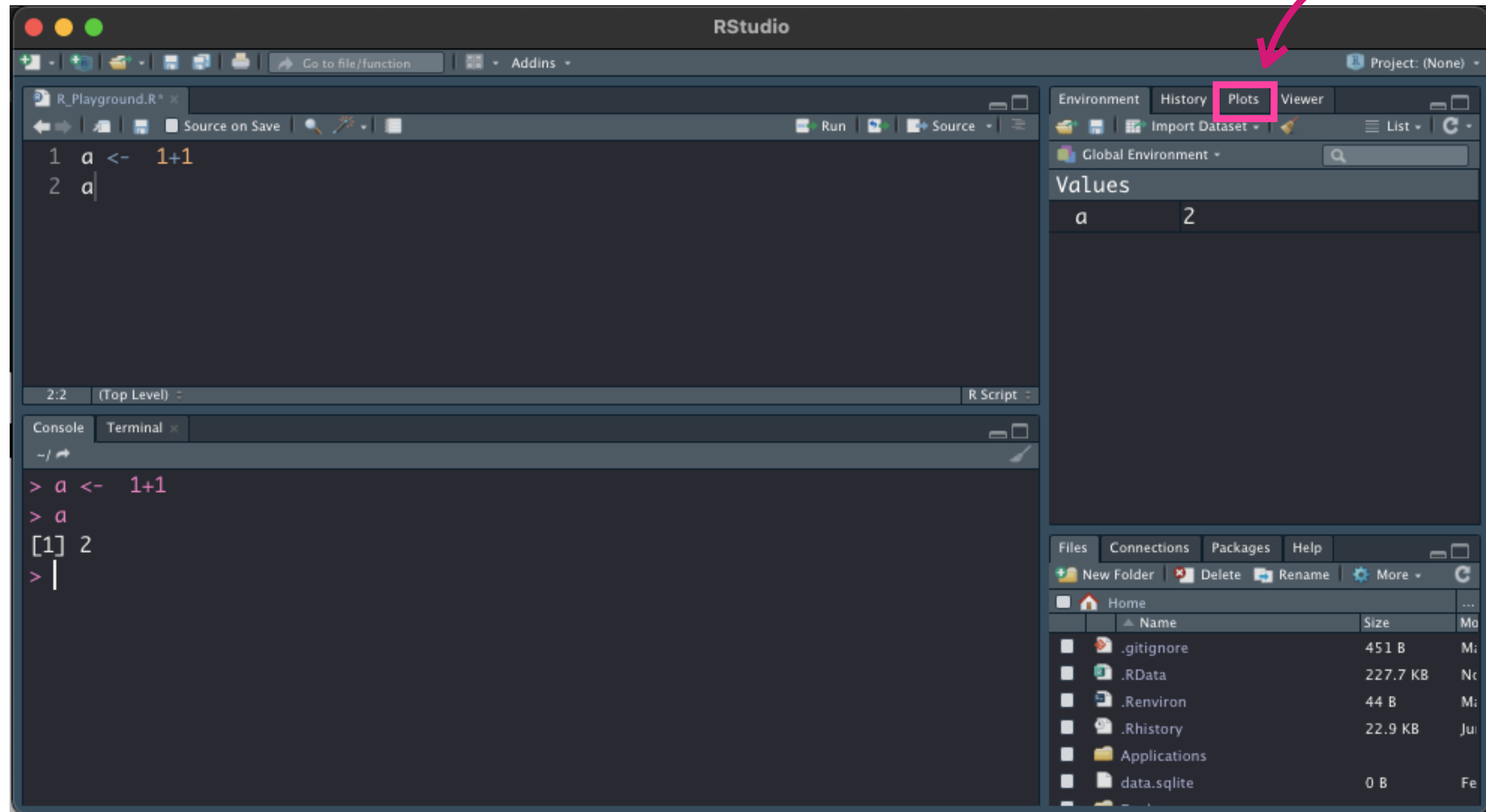
Abbildung 3
R-Studio IDE



Environment:
Übersicht über
alle Variablen im
Speicher

R-Studio

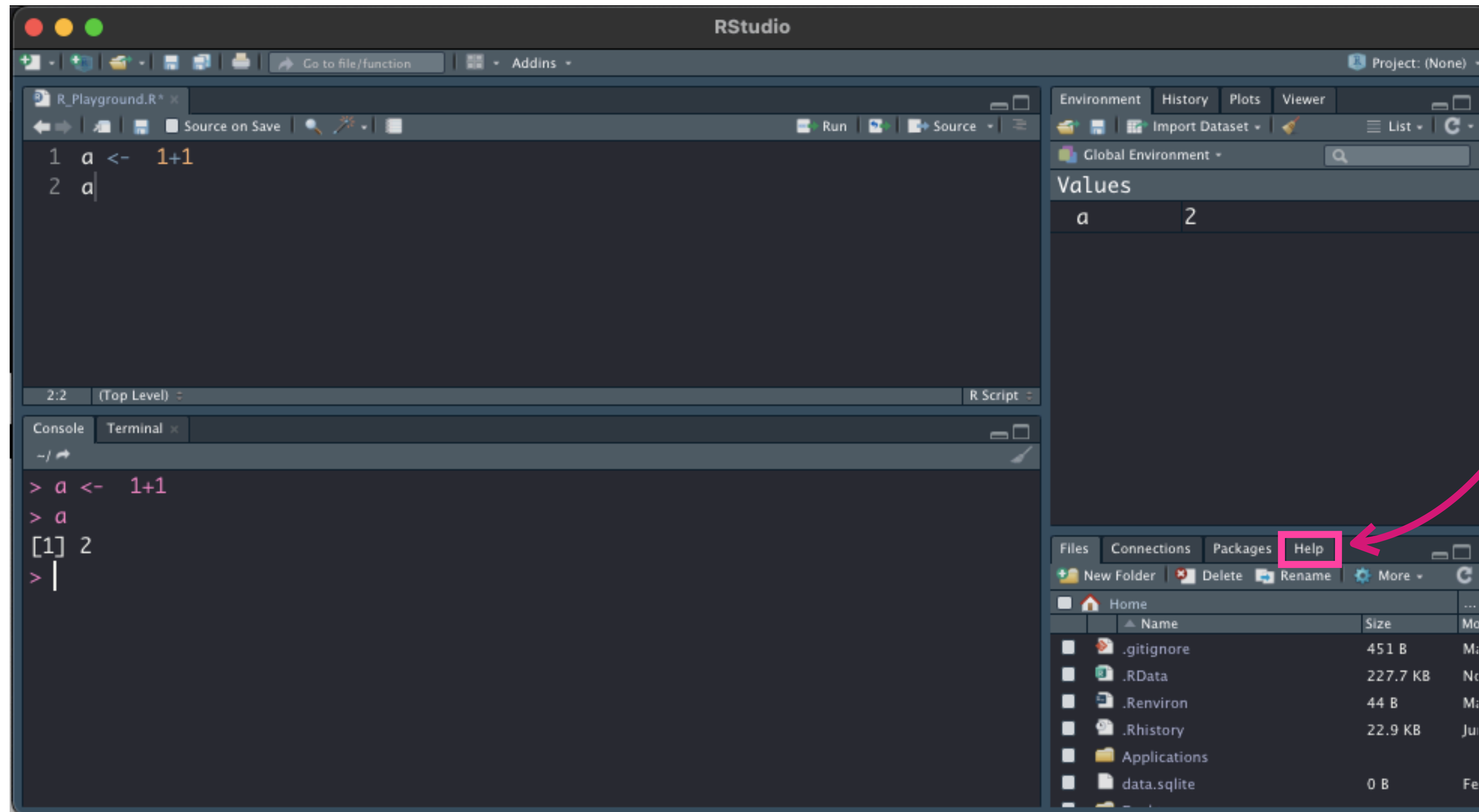
Abbildung 3
R-Studio IDE



Plots:
Zeigt euch eure
Abbildungen an

R-Studio

Abbildung 3
R-Studio IDE



Help:
z.B. Package
Dokumentation
anschauen

R-ste Schritte

Arten von Daten:

Character Strings (Verkettungen von Zeichen): „abc“, „Das ist ein Satz“, „ab39gHb“, „2783“, „:-)“

Integers / Numerics (diskrete Werte = Zahlen): 42, 82736.6666

Booleans / Logicals (boolsche/binäre Ausdrücke): TRUE, FALSE, T, F

Fehlende Werte: NA

Factors: kategoriale Werte, die eine bestimmte Ordnung haben; Level werden als Character gespeichert

Datentypen verändern (= typecasten):

Beispiel: Numeric in Character umwandeln

```
as.character(1234)
```

```
> [1] „1234“
```

```
as.numeric(„1234“)
```

```
> [1] 1234
```

Achtung! Beim Typecasten von factors zu numerics muss man aufpassen, weil sonst ggf. die Werte verändert werden.

Nutzt am besten einen der folgenden Befehle:

```
as.numeric(levels(f))[f]
```

```
as.numeric(as.character(f))
```


R-ste Schritte

Daten organisieren:

Einzelwerte als Objekte anlegen:

```
a <- 17
```

Wenn man dann a ausführt, wird der Wert 17 ausgegeben:

```
> [1] 17
```

Vektor anlegen:

```
a <- c(1,2,3)
```

Wenn man dann a ausführt, wird der Vektor ausgegeben:


```
> [1] 1 2 3
```

Matrix erstellen:

mehrere Vektoren als Zeilen aneinander binden:

```
z <- rbind(vektor1, vektor2)
```

View(z)



vektor1	1	2	3	4
vektor2	10	16	8	9

Matrix erstellen:

mehrere Vektoren als Spalten aneinander binden:

```
z <- cbind(vektor1, vektor2)
```

View(z)



vektor1	vektor2
1	10
2	16
3	8

Matrix in einen Dataframe umwandeln:

```
my_df <- as.data.frame(z)
```

R-ste Schritte

Rechenoperationen:

Addieren: $a + b$

Subtrahieren: $a - b$

Multiplizieren: $a * b$

Dividieren: a / b

Potenz: a^b

Wurzel ziehen: `sqrt(a)`

Mittelwert: `mean(vektor)`

Median: `median(vektor)`

Standardabweichung: `sd(vektor)`

Minimum und Maximum in einem Vektor finden: `range(vektor)`

Anzahl von Elementen in einem Vektor zählen: `length(vektor)`

Funktionen

Eine Gärtnerin pflanzt Mango- und Limettenbäume. Sie weiß, dass jeder Mangobaum im Schnitt 21 Früchte tragen wird und jeder Limettenbaum 100. Sie möchte wissen, wie viele Früchte sie insgesamt ernten wird.

eigene Funktion schreiben

```
comp_harvest <- function(mangotrees, limetrees){ # Argumente definieren
  # mit Argumenten rechnen:
  mangos <- mangotrees * 21
  limes <- limetrees * 100
  harvest <- mangos + limes
  # neuen Wert ausgeben lassen:
  return(harvest)
}
```

Funktionen

Eine Gärtnerin pflanzt Mango- und Limettenbäume. Sie weiß, dass jeder Mangobaum im Schnitt 21 Früchte tragen wird und jeder Limettenbaum 100. Sie möchte wissen, wie viele Früchte sie insgesamt ernten wird.

eigene Funktion schreiben

```
comp_harvest <- function(mangotrees, limetrees){ # Argumente definieren
  # mit Argumenten rechnen:
  mangos <- mangotrees * 21
  limes <- limetrees * 100
  harvest <- mangos + limes
  # neuen Wert ausgeben lassen:
  return(harvest)
}
```

eigene Funktion benutzen:

```
comp_harvest(mangotrees = 2, limetrees = 3) # 2 * 21 Mangos und 3 * 100 Limetten = 342 Früchte insg.
> [1] 342
```

Argumentnamen müssen nicht genannt werden, nur die richtige Reihenfolge ist wichtig:

```
comp_harvest(2, 3)
> [1] 342
```


Funktionen

Optionale Argumente in Funktionen

Die Gärtnerin weiß, dass nur gesunde Limettenbäume Früchte tragen. Sie geht aber grundsätzlich davon aus, dass alle ihre Bäume gesund sind, sofern sie keinen kranken Baum findet.

```
comp_harvest <- function(mangotrees, limetrees, sick_limetrees = 0){ # Default-Wert für Zusatzargument
  # normal mit Argumenten rechnen:
  mangos <- mangotrees * 21
  limes <- (limetrees - sick_limetrees) * 100
  harvest <- mangos + limes
  # neuen Wert ausgeben lassen:
  return(harvest)
}
```

Funktionen

Optionale Argumente in Funktionen

Die Gärtnerin weiß, dass nur gesunde Limettenbäume Früchte tragen. Sie geht aber grundsätzlich davon aus, dass alle ihre Bäume gesund sind, sofern sie keinen kranken Baum findet.

```
comp_harvest <- function(mangotrees, limetrees, sick_limetrees = 0){ # Default-Wert für Zusatzargument
  # normal mit Argumenten rechnen:
  mangos <- mangotrees * 21
  limes <- (limetrees - sick_limetrees) * 100
  harvest <- mangos + limes
  # neuen Wert ausgeben lassen:
  return(harvest)
}
```

Funktion ohne Zusatzargument benutzen:

```
comp_harvest(mangotrees = 2, limetrees = 3) # kein Baum krank, nach wie vor 342 Früchte insg.
> [1] 342
```

Funktion mit Zusatzargument benutzen:

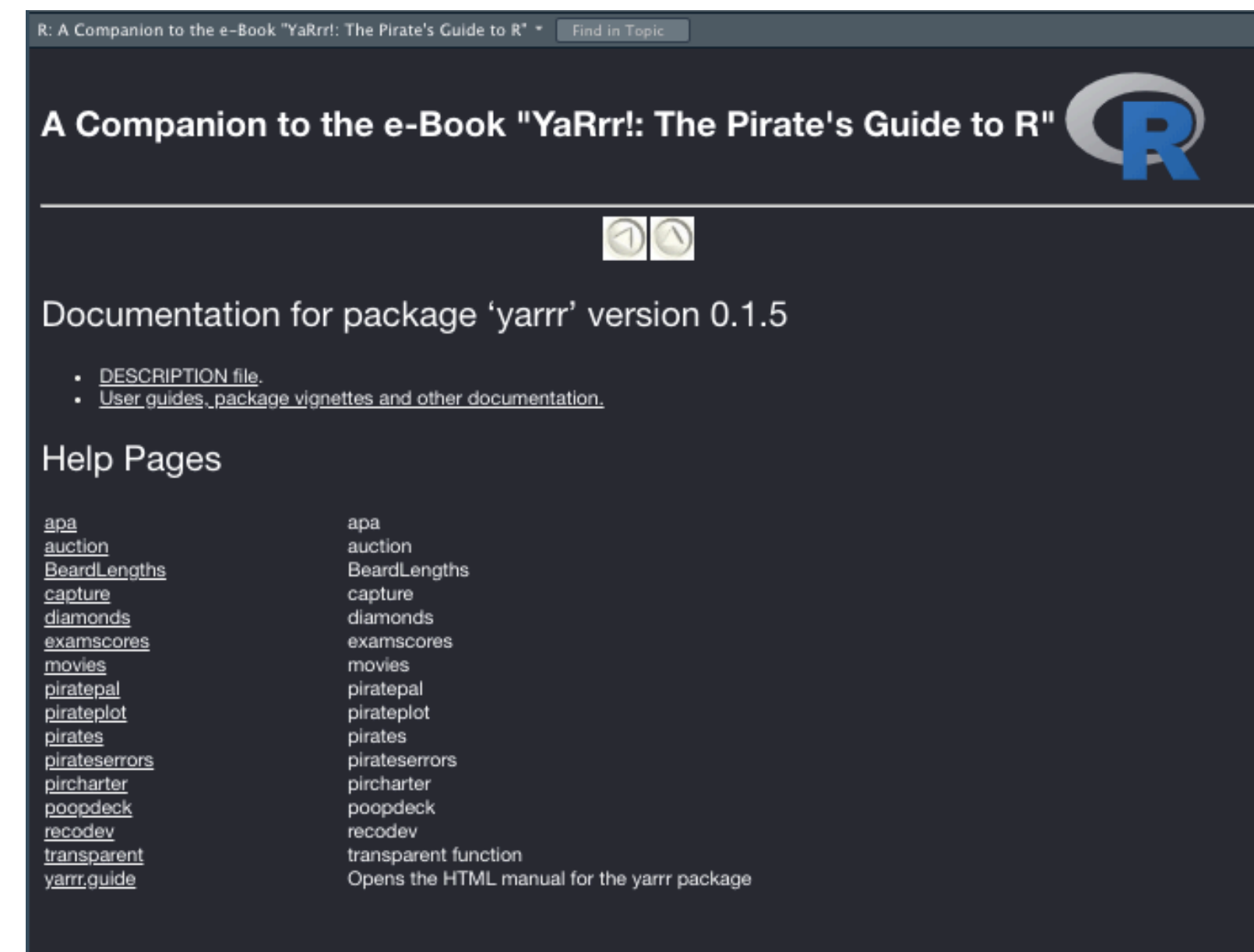
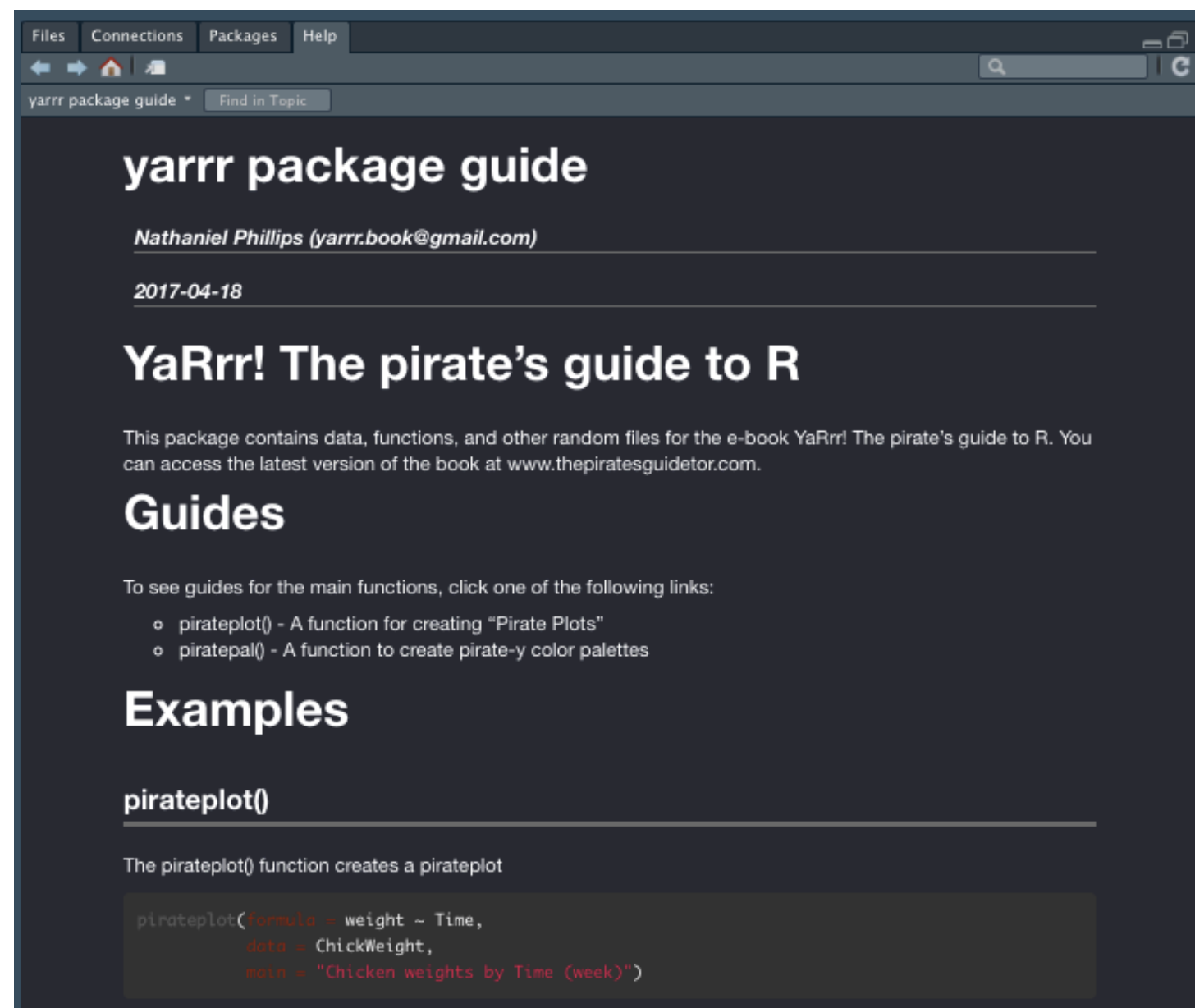
```
comp_harvest(mangotrees = 2, limetrees = 3, sick_limetrees = 1) # 1 Baum krank = 100 Früchte weniger
> [1] 242
```

Funktionen

Funktionen muss man nicht immer selbst schreiben, man kann auch packages mit fertigen Funktionen importieren:

```
install.packages(„yarr“) # package installieren  
library(yarr) # package laden
```

```
yarr.guide() # optional bei einigen Packages: package guide anzeigen lassen  
help(package = yarr)
```



if-Loops

Nutzt man, wenn man Teile seines Codes nur ausführen will, wenn eine bestimmte Bedingung gegeben ist (oder mehrere Bedingungen).

```
a <- 1  
b <- 2
```

```
if (a == b) {  
  c <- 1
```

```
} else if (a < b) {  
  c <- 2
```

```
} else { # if a > b  
  c <- 3
```

```
}
```

Was ist c?

if-Loops

Nutzt man, wenn man Teile seines Codes nur ausführen will, wenn eine bestimmte Bedingung gegeben ist (oder mehrere Bedingungen).

```
a <- 1  
b <- 2
```

```
if (a == b) {  
  c <- 1
```

```
} else if (a < b) {  
  c <- 2
```

```
} else { # if a > b  
  c <- 3
```

```
}
```

Was ist c? **Richtige Antwort: 2**

Logische Operatoren:

a ist gleich b: $a == b$

a ist ungleich b: $a != b$

a ist größer als b: $a > b$

a ist größer gleich b: $a >= b$

a ist kleiner als b: $a < b$

a ist kleiner gleich b: $a <= b$

a ist gleich b ODER c:

$a == b \mid a == c$

a ist gleich b UND c:

$a == b \& a == c$

a ist ein NA: `is.na(a)`

A ist kein NA: `!is.na(a)`

if-Loops

Nutzt man, wenn man Teile seines Codes nur ausführen will, wenn eine bestimmte Bedingung gegeben ist (oder mehrere Bedingungen).

Beispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Beispieldatensatz *df*:

	code	seeing_mice	gender	age	mean_RT	exclude
1	VP1	FALSE	female	23	300.56	NA
2	VP2	TRUE	male	21	800.45	NA
3	VP3	FALSE	female	20	376.45	NA
4	VP4	FALSE	nonbinary	22	342.82	NA

if-Loops

Nutzt man, wenn man Teile seines Codes nur ausführen will, wenn eine bestimmte Bedingung gegeben ist (oder mehrere Bedingungen).

Beispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Beispieldatensatz „df“:

	code	seeing mice	gender	age	mean RT	exclude
1	VP1	FALSE	female	23	300.56	FALSE
2	VP2	TRUE	male	21	800.45	NA
3	VP3	FALSE	female	20	376.45	NA
4	VP4	FALSE	nonbinary	22	342.82	NA

```
if (df$seeing_mice[1] == TRUE){  
  df$exclude[1] <- TRUE  
} else {  
  df$exclude[1] <- FALSE  
}
```

if-Loops

Nutzt man, wenn man Teile seines Codes nur ausführen will, wenn eine bestimmte Bedingung gegeben ist (oder mehrere Bedingungen).

Beispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Beispieldatensatz „df“:

	code	seeing_mice	gender	age	mean_RT	exclude
1	VP1	FALSE	female	23	300.56	FALSE
2	VP2	TRUE	male	21	800.45	TRUE
3	VP3	FALSE	female	20	376.45	NA
4	VP4	FALSE	nonbinary	22	342.82	NA

```
if (df$seeing_mice[2] == TRUE){  
  df$exclude[2] <- TRUE  
} else {  
  df$exclude[2] <- FALSE  
}
```


if-Loops

Nutzt man, wenn man Teile seines Codes nur ausführen will, wenn eine bestimmte Bedingung gegeben ist (oder mehrere Bedingungen).

Beispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Beispieldatensatz „df“:

	code	seeing_mice	gender	age	mean_RT	exclude
1	VP1	FALSE	female	23	300.56	FALSE
2	VP2	TRUE	male	21	800.45	TRUE
3	VP3	FALSE	female	20	376.45	FALSE
4	VP4	FALSE	nonbinary	22	342.82	NA

```
if (df$seeing_mice[3] == TRUE){  
  df$exclude[3] <- TRUE  
} else {  
  df$exclude[3] <- FALSE  
}
```

if-Loops

Nutzt man, wenn man Teile seines Codes nur ausführen will, wenn eine bestimmte Bedingung gegeben ist (oder mehrere Bedingungen).

Beispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Beispieldatensatz „df“:

	code	seeing_mice	gender	age	mean_RT	exclude
1	VP1	FALSE	female	23	300.56	FALSE
2	VP2	TRUE	male	21	800.45	TRUE
3	VP3	FALSE	female	20	376.45	FALSE
4	VP4	FALSE	nonbinary	22	342.82	FALSE

```
if (df$seeing_mice[4] == TRUE){  
  df$exclude[4] <- TRUE  
} else {  
  df$exclude[4] <- FALSE  
}
```

for-Loops

Nutzt man, wenn man Teile seines Codes eine bestimmte Anzahl von Malen wiederholen möchte (quasi Alternative zu Copy & Pasten von Code).

Beispiel: Ich möchte 10 Zufallszahlen „würfeln“ und die einzeln mit der print-Funktion ausgeben lassen.

Option 1:
Code 10x wiederholen

```
print( floor(runif(1,1,7)) )  
print( floor(runif(1,1,7)) )  
print( floor(runif(1,1,7)) )  
print( floor(runif(1,1,7)) )  
print( floor(runif(1,1,7)) )  
print( floor(runif(1,1,7)) )  
print( floor(runif(1,1,7)) )  
print( floor(runif(1,1,7)) )  
print( floor(runif(1,1,7)) )  
print( floor(runif(1,1,7)) )
```

Option 2:
for-Loop mit 10 Durchgängen

```
for (i in 1:10){  
  print(floor(runif(1,1,7)))  
}
```

Option 3:
10 Zufallszahlen würfeln und mit for-Loop einzeln ausgeben lassen

```
random_num <- floor(runif(10,1,7))  
  
for (i in random_num){  
  print(i)  
}
```

Dieser Codeschnipsel
generiert eine
ganzzahlige Zufallszahl
zwischen 1 und 6



for-Loops

Anwendungsbeispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Zusatz: Ich möchte aber nicht per Hand die Indizes in den Code einsetzen und ihn immer wieder ausführen.

Beispieldatensatz „df“:

	code	seeing_mice	gender	age	mean_RT	exclude
1	VP1	FALSE	female	23	300.56	NA
2	VP2	TRUE	male	21	800.45	NA
3	VP3	FALSE	female	20	376.45	NA
4	VP4	FALSE	nonbinary	22	342.82	NA

```
if (df$seeing_mice[1] == TRUE){  
  df$exclude[1] <- TRUE  
} else {  
  df$exclude[1] <- FALSE  
}
```

for-Loops

Anwendungsbeispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Zusatz: Ich möchte aber nicht per Hand die Indizes in den Code einsetzen und ihn immer wieder ausführen.

```
for ( i in 1:length(df$seeing_mice) ){ # für jede Zeile im Datensatz df...
```

```
  # führe unseren Codeschnipsel aus:
```

```
  if (df$seeing_mice[i] == TRUE){
```

```
    df$exclude[i] <- TRUE
```

```
  } else {
```

```
    df$exclude[i] <- FALSE
```

```
  }
```

```
}
```



mit i als „Platzhalter“ für
den Index, den wir
sonst immer per Hand
ändern müssten

while-Loops

Anwendungsbeispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Ich möchte aber nicht per Hand die Indices in den Code einsetzen und ihn immer wieder ausführen.

Zusatz: Ich möchte den Code nur so lange ausführen, bis ich die eine VP gefunden habe, die rausgeschmissen werden soll, weil ich weiß, dass es nur eine gibt.

Beispieldatensatz „df“ vor dem Loop:

	code	seeing_mice	gender	age	mean_RT	exclude
1	VP1	FALSE	female	23	300.56	NA
2	VP2	TRUE	male	21	800.45	NA
3	VP3	FALSE	female	20	376.45	NA
4	VP4	FALSE	nonbinary	22	342.82	NA

Beispieldatensatz „df“ nach dem while-Loop:

	code	seeing_mice	gender	age	mean_RT	exclude
1	VP1	FALSE	female	23	300.56	FALSE
2	VP2	TRUE	male	21	800.45	TRUE
3	VP3	FALSE	female	20	376.45	NA
4	VP4	FALSE	nonbinary	22	342.82	NA

while-Loops

Anwendungsbeispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Ich möchte aber nicht per Hand die Indices in den Code einsetzen und ihn immer wieder ausführen.

Zusatz: Ich möchte den Code nur so lange ausführen, bis ich die eine VP gefunden habe, die rausgeschmissen werden soll.

```
i <- 1
found_subject <- FALSE

while ( found_subject == FALSE){ # solange die VP nicht gefunden wurde...

  # führe unseren Codeschnipsel aus:
  if (df$seeing_mice[i] == TRUE){
    df$exclude[i] <- TRUE
    found_subject <- TRUE # VP gefunden, beende Loop!
  } else {
    df$exclude[i] <- FALSE
    i = i + 1 # gehe zur nächsten Zeile
  }

}
```

while-Loops

Anwendungsbeispiel: Ich möchte alle VPn ausschließen, die angegeben haben, dass sie immer überall weiße Mäuse sehen, die anderen sollen nicht ausgeschlossen werden.

Ich möchte aber nicht per Hand die Indices in den Code einsetzen und ihn immer wieder ausführen.

Zusatz: Ich möchte den Code nur so lange ausführen, bis ich die eine VP gefunden habe, die rausgeschmissen werden soll.

```
i <- 1
found_subject <- FALSE

while ( found_subject == FALSE){ # solange die VP nicht gefunden wurde...
```

```
  # führe unseren Codeschnipsel aus:
```

```
  if (df$seeing_mice[i] == TRUE){
```

```
    df$exclude[i] <- TRUE
```

```
    found_subject <- TRUE # VP gefunden, beende Loop!
```

```
  } else {
```

```
    df$exclude[i] <- FALSE
```

```
    i = i + 1 # gehe zur nächsten Zeile
```

```
  }
```

```
}
```

Wichtig:
Bei while-Loops
immer an die
Abbruchbedingung
denken!

for-Loops

```
for ( i in 1:length(df$seeing_mice) ){ # für jede Zeile im Datensatz df...
```

```
# führe unseren Codeschnipsel aus:
```

```
if (df$seeing_mice[i] == TRUE){
```

```
  df$exclude[i] <- TRUE
```

```
  break
```

```
} else {
```

```
  df$exclude[i] <- FALSE
```

```
}
```

```
}
```

Ihr könnt einen Loop auch abbrechen, indem ihr statt einer Abbruchbedingung einfach *break* schreibt.

Analyseskript

Plan für heute:

- Daten einlesen

- wichtige Daten aus den Datensätzen ziehen und bereinigen

- Deskriptive Statistik

- Inferenzstatistik

- Plot bauen

Das fertige Skript findet ihr auf Github unter

https://github.com/MMarieSchuckart/R_public/blob/main/R_Tutorial_June2021

Abbildungsverzeichnis

Abbildung 1: Imgflip. (n.D.). *Grandma finds the Internet* [Fotografie]. Abgerufen von <https://imgflip.com/memegenerator/Grandma-Finds-The-Internet>

Abbildung 2 & 4: IStock. (2017). *Distracted Boyfriend* [Fotografie]. Abgerufen von <https://knowyourmeme.com/memes/distracted-boyfriend>

Abbildung 3: Imgflip. (2019). *X-X-Everywhere* [Filmausschnitt]. Abgerufen von <https://imgflip.com/memetemplate/X-X-Everywhere>