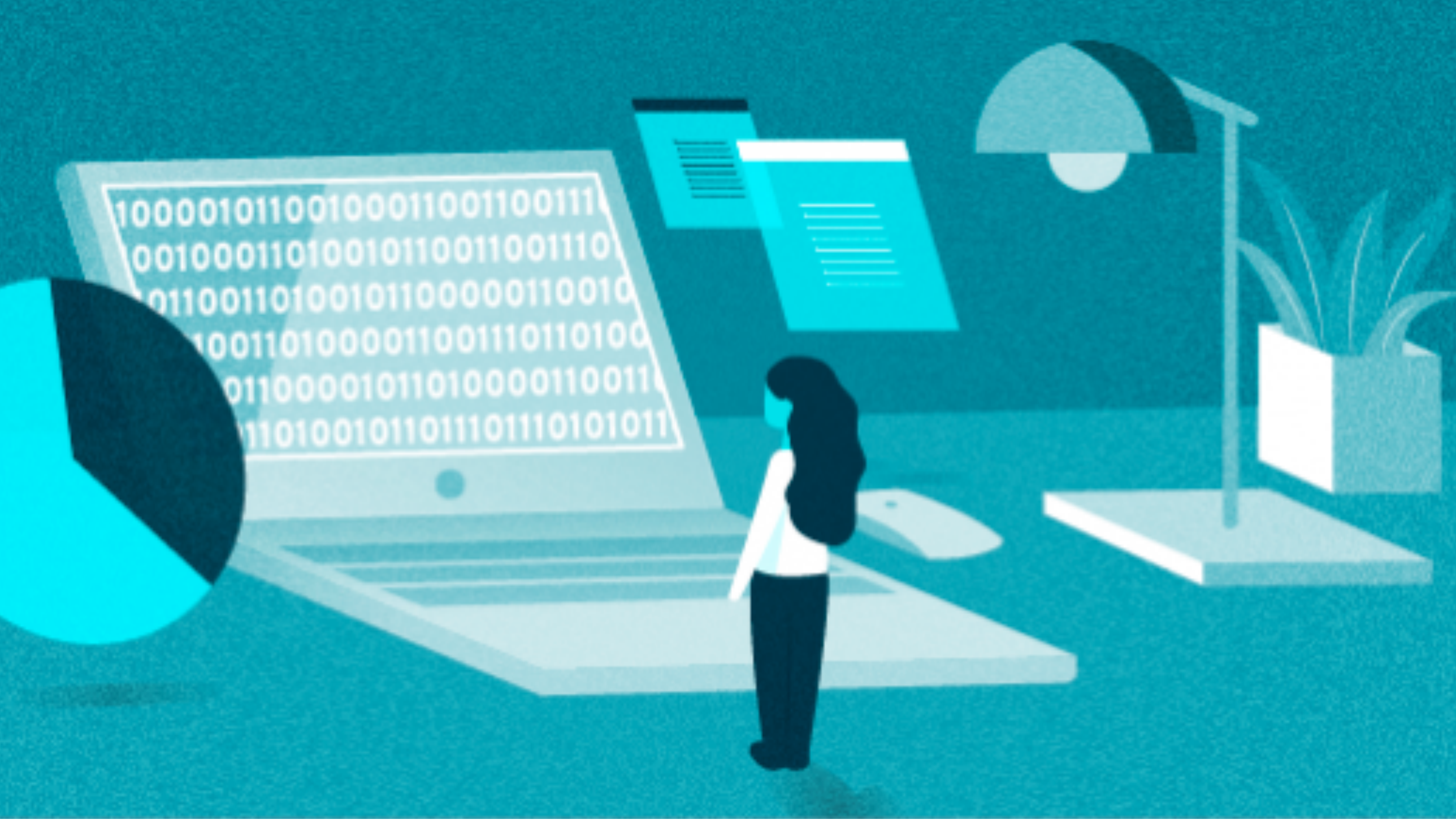


MAKING ONLINE EXPERIMENTS

A/Prof Amy Perfors
University of Melbourne
CoEDL Summer School 2019

PLAN FOR THE COURSE

Ambitious — there is no way you will become an expert in running experiments online after six hours. My goal is to provide you with enough basics and pointers to resources that you'll be able to do the rest yourself

TODAY

1. Basics of Javascript/jsPsych
2. Using the R package jaysire to interface with jsPsych and create a simple experiment that runs locally

TOMORROW

1. Put our simple experiment from yesterday onto Google App Engine, and download the data
2. Add some cool things to our experiment

THE BASIC IDEA

Server side

In this case, this is you! You (or your server, which will be Google App Engine) are serving up an experiment to your participant.

Client side

The client's web browser serves up webpages.

As programmer, you are writing code so that the browser on the client side knows what to do

Usually **HTML**: a markup language for displaying all your content

Javascript is a client-side language that lets you do more complex things. It is embedded in html. I'll be talking about a particular library called **jsPsych** used for making online experiments

Jaysire is a **R package** consisting of wrapper functions for javascript. It means you can write the code in R and it will translate to javascript for you

THE BASIC IDEA

Statistical programming language

- Do calculations
- Draw figures
- Do statistical analyses
- Write programs

- Make webpages
- Write books
- Write manuscripts
- Interface with jsPsych

RStudio

IDE (interface) for R

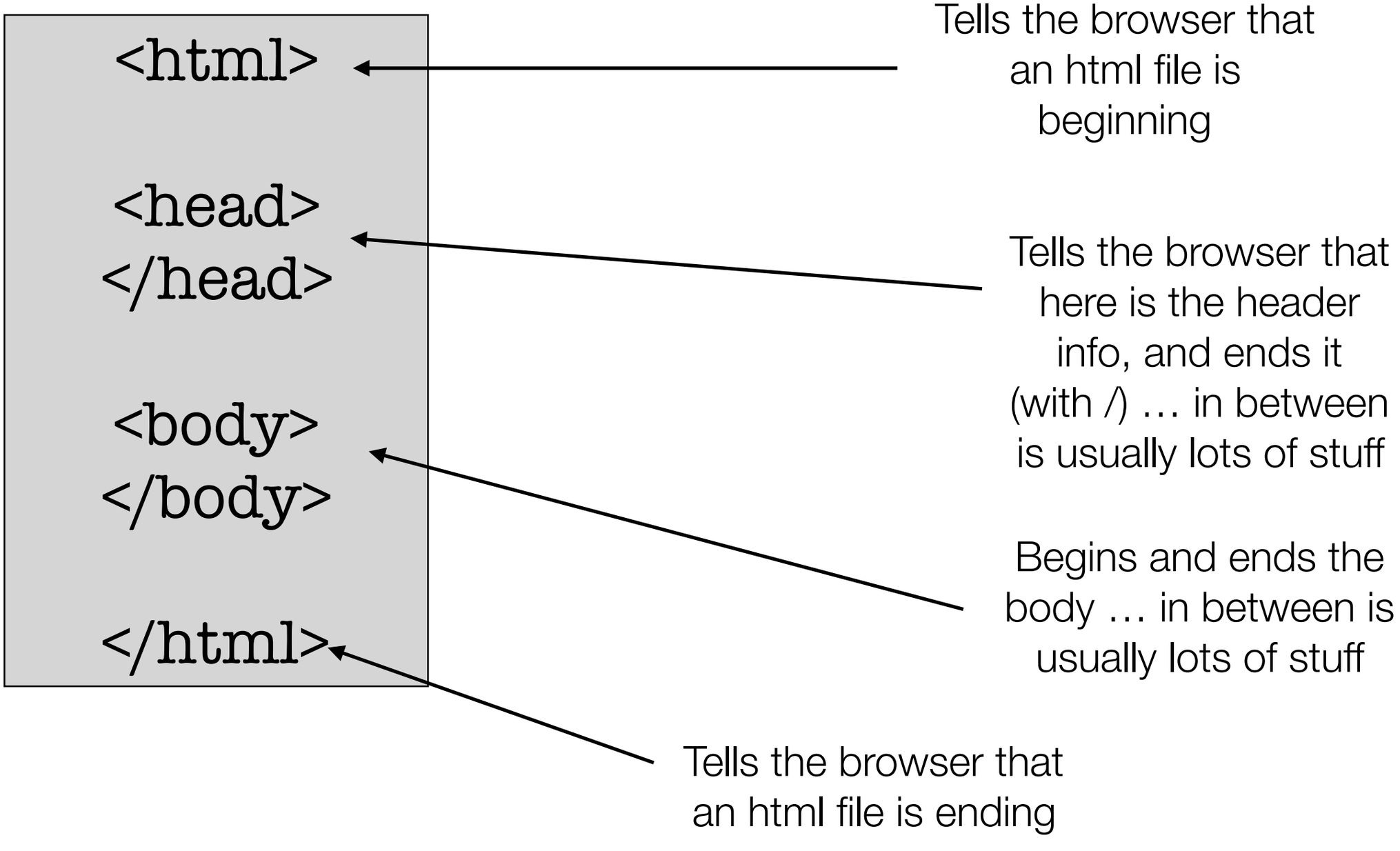
- Use it!
- GUIs, manages projects

- Can also use it for html, javascript, etc
- We'll do everything in RStudio

(quick demo)

HTML

Any webpage has the same basic structure



HTML

HTML works by using tags, which are basically commands for the browers. Many lists of tags can be found online, e.g. here: https://www.w3schools.com/tags/

<a> ← Hyperlink

```
<a href = “http://chdsummerschool.com">Link to summer school</a>
```

**
** ← Line break

```
Hello!<br>I am so pleased to be here.
```

Hello!
I am so pleased to be here.

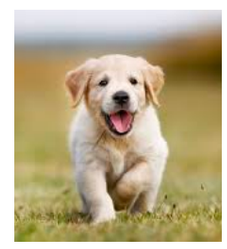
**** ← Emphasised text

```
Summer school is <em>so</em> cool.
```

Summer school is *so* cool.

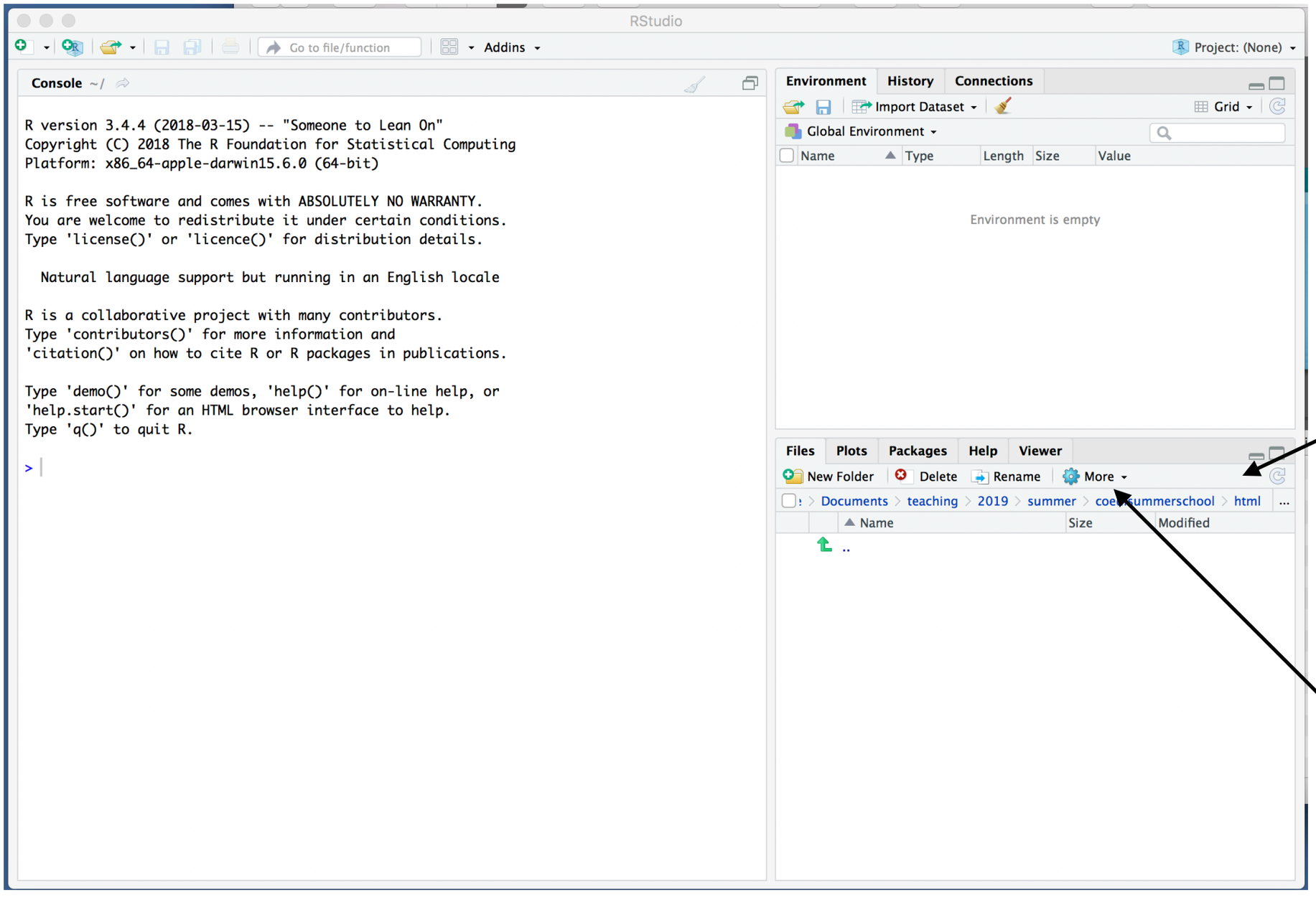
<img> ← Defines an image

```
<img src=“puppy.jpg” alt=“cutepuppy”>
```



HTML

Let's make a super basic webpage.

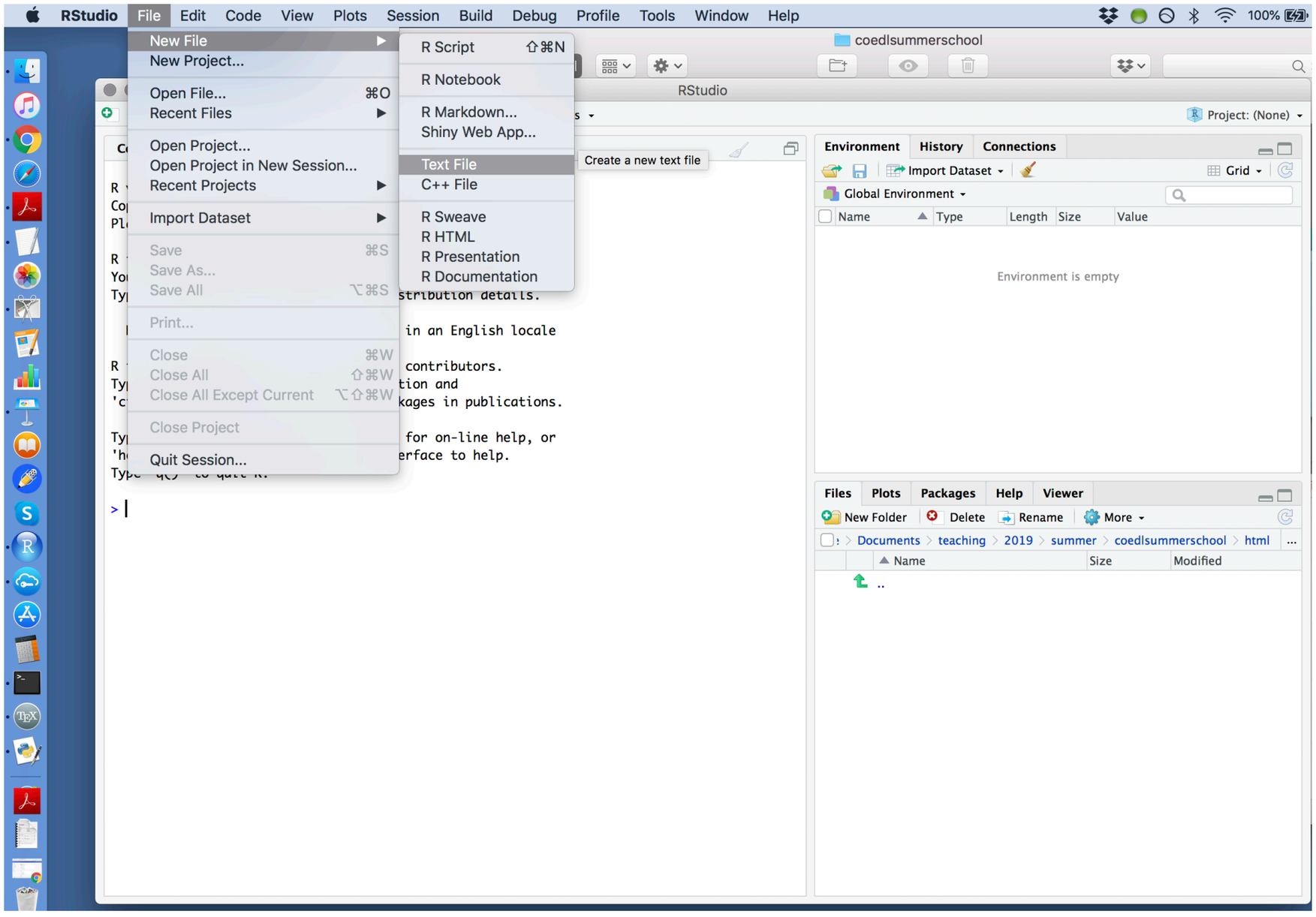


Open RStudio, and navigate to a new folder

Set it as your working directory

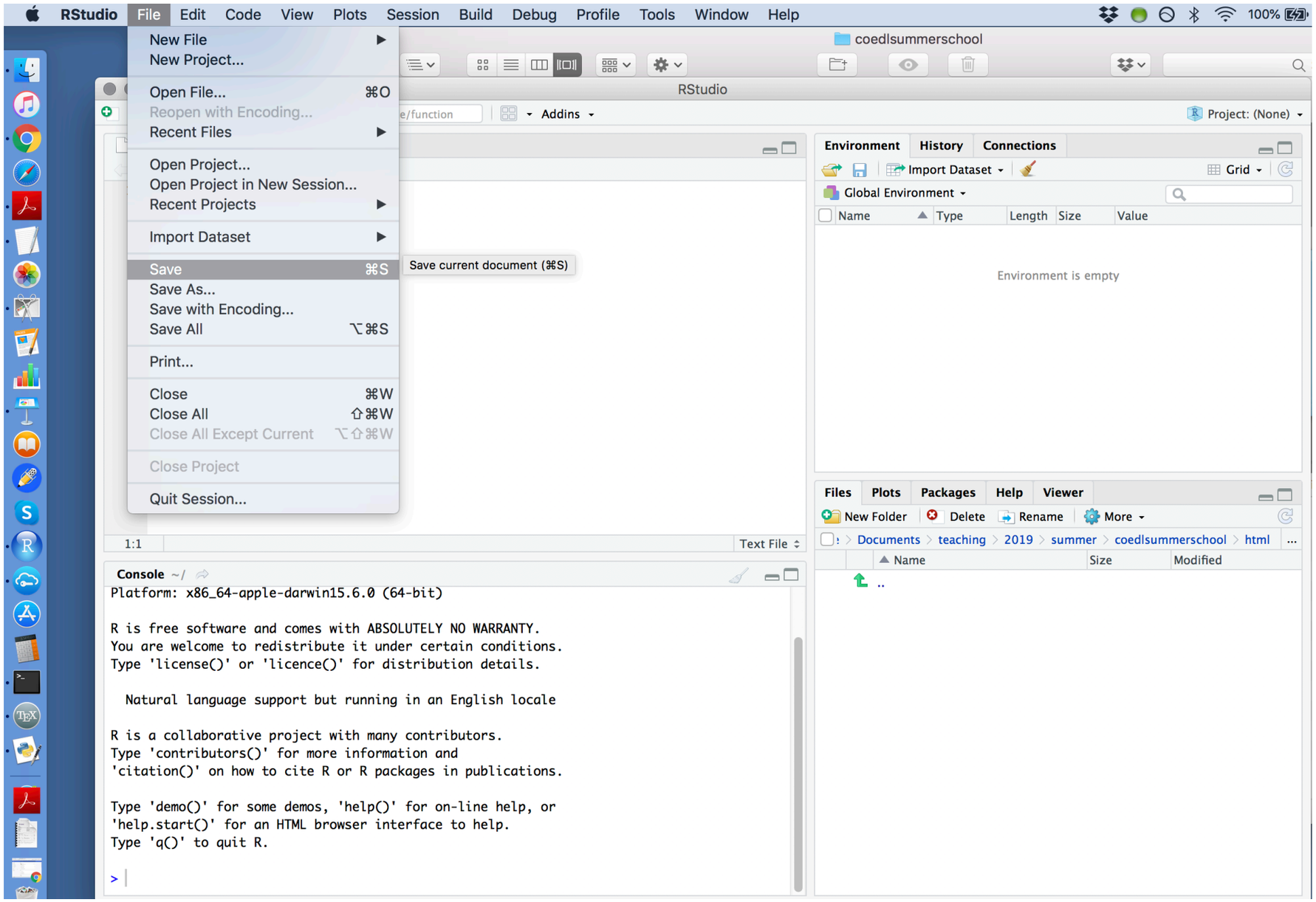
HTML

Make a new file (choose ‘text file’. R HTML would work but that adds a bunch of stuff we don’t need right now)



HTML

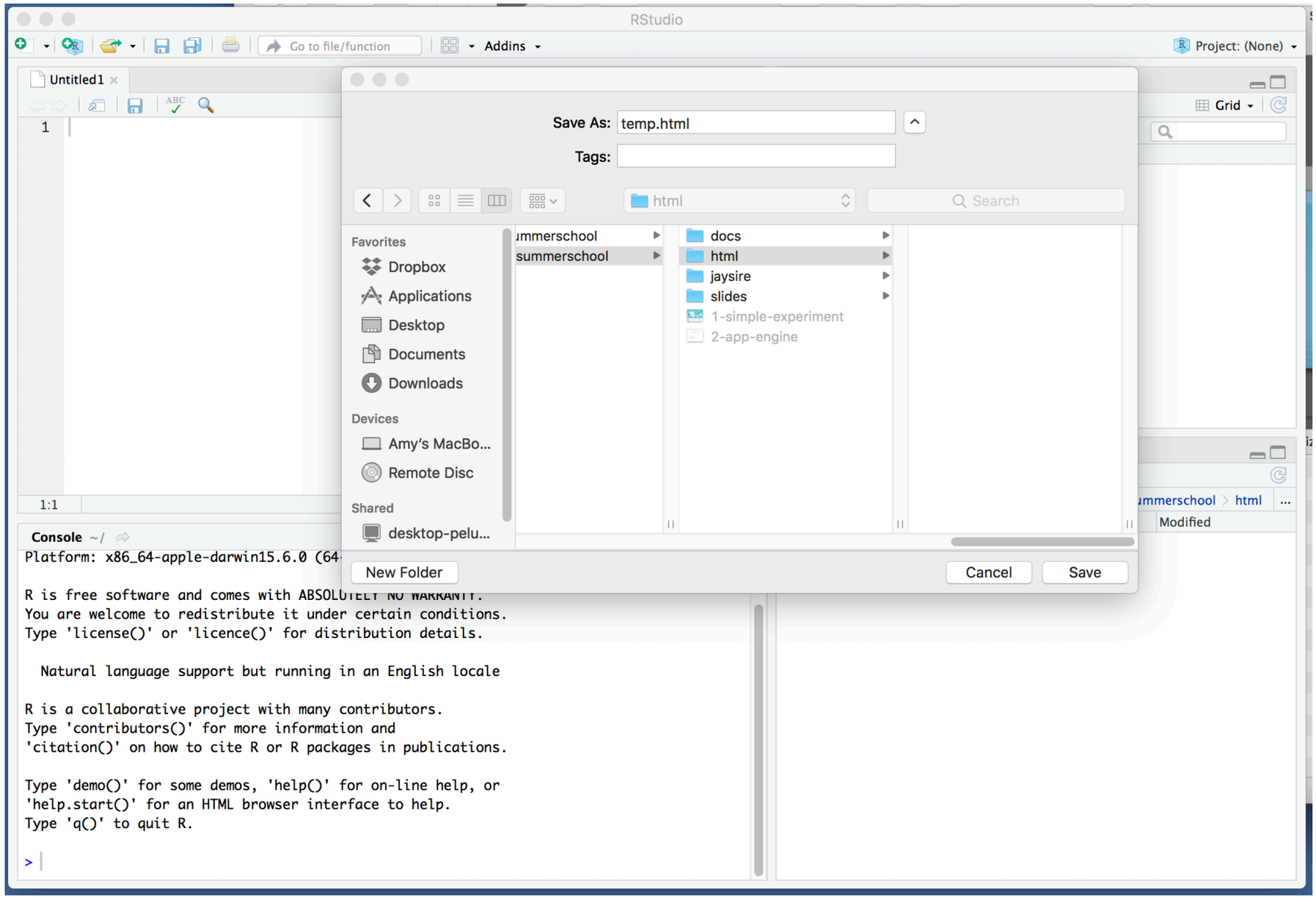
Make a new file (choose ‘text file’. R HTML would work but that adds a bunch of stuff we don’t need right now)



Because I’m paranoid, I always save first thing

HTML

Make a new file (choose ‘text file’. R HTML would work but that adds a bunch of stuff we don’t need right now)



Because I’m paranoid, I always save first thing

HTML

Now let's create a hello message using our template.

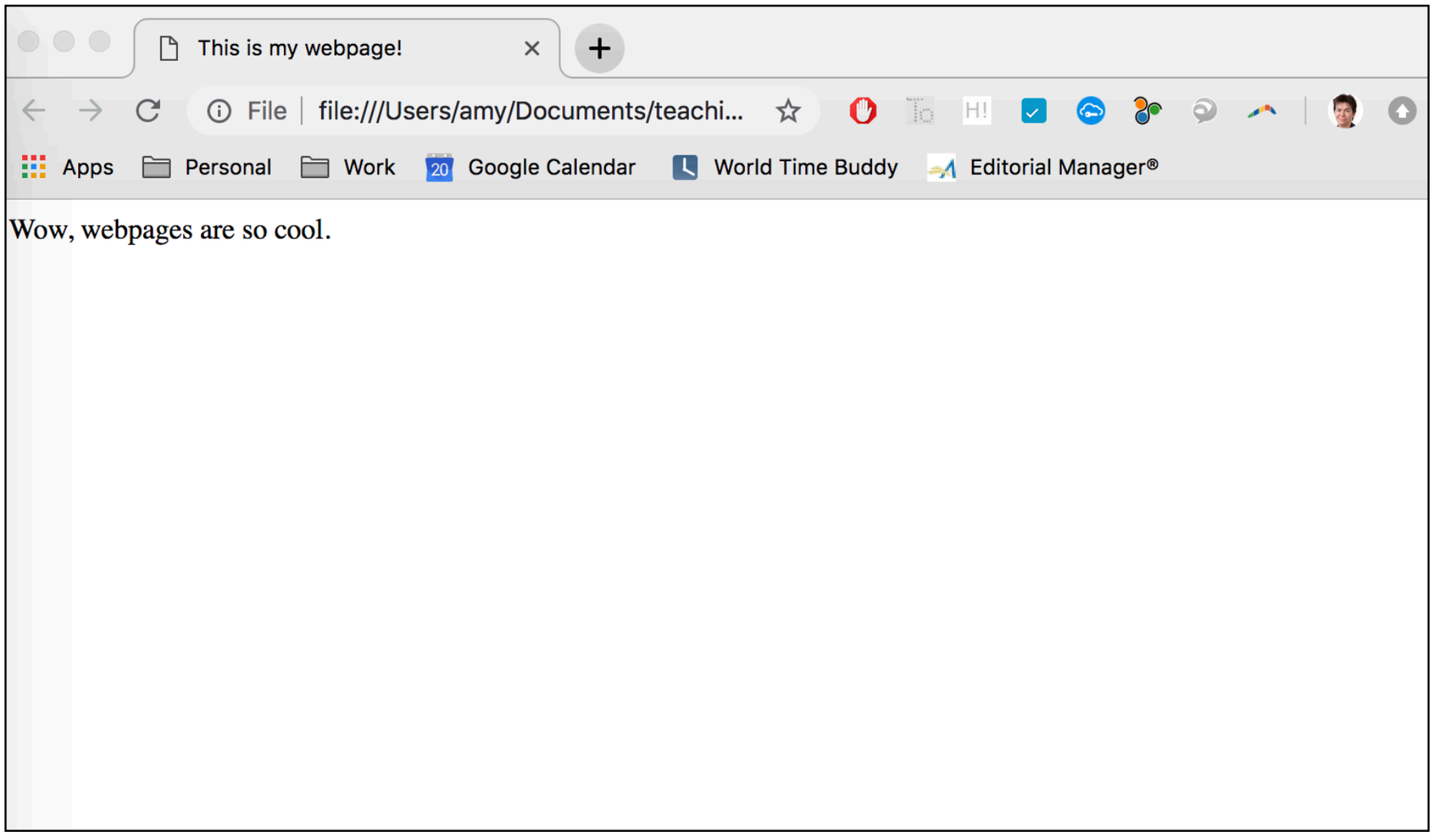
temp.html

Preview

```
1  <html>
2    <head>
3      <title>This is my webpage!</title>
4    </head>
5    <body>
6      Wow, webpages are so cool.
7    </body>
8
9  </html>
```

HTML

Save and click on **`temp.html`** and you should see something like this come up in your browser!



HTML

Let's add a few more useful tags…

<p>

Paragraph break

Wow!<p>This is so cool.

Wow!

This is so cool.

,

List making

```
<ol>
   <li> Pianos </li>
  <li> Lettuce </li>
  <li> Cockatoos </li>
</ol>
```

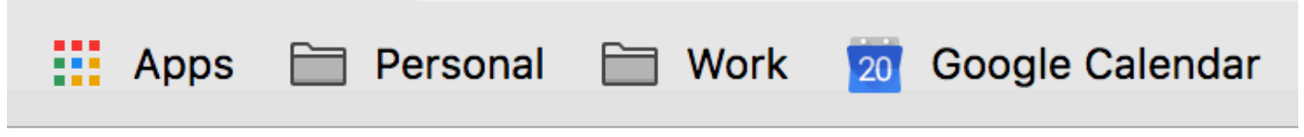
1. Pianos
2. Lettuce
3. Cockatoos

Defines a section ("division") of your page which you can refer to later

```
<ol>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ol>
```

EXERCISES

Modify `temp.html` so the website looks like the one below:



Hello world!

My favourite things are:

1. Bunnies.
2. Even more bunnies.
3. Minecraft.

Here is a link to the summer school website.

HOW DOES JAVASCRIPT COME IN?

Remember that javascript is *code* that can go into a webpage so it can do more complicated things than display information. There are lots of ways to do this, but we'll start with a very simple exercise to illustrate the idea.

HOW DOES JAVASCRIPT COME IN?

First thing we want to do is put the actual functions there; they can go in the head or after the body.

```
</body>
<script language="JavaScript">
  function temperature(form) {
    var c = parseFloat(form.DegC.value, 10);
    var f = 0;
    f = c * (9.0/5.0) + 32;
    form.DegF.value = f;
  }
</script>
```

Tells the browser what language the code is in; this is optional, can just do `<script>`

Name of the function is `temperature` and it takes a `form` element as input

Two variables: `c` is the value entered at the form, `f` is what we are converting to

Once we calculate the temp in F, we assign that to the relevant value on the form

HOW DOES JAVASCRIPT COME IN?

This by itself does nothing visible, because it doesn't affect what is showing on the webpage. For that we need to change the body.

```
<body>
  <form>
    <div class="question" style="width: 1000px">
      <h2>Celsius to Fahrenheit Converter</h2>
      Enter a temperature in degrees C:
      <input name="DegC" value="0" maxlength="15" size=15>
      <br>
    </div>
    <div class="answer" style="width: 1000px">
      Click this button to calculate the temperature in degrees F:
      <input name="calc" value="Calculate" type="button" onClick=temperature(this.form)>
      <br><br>
      Temperature in degrees F is:
      <input name="DegF" readonly size=15>
    </div>
  </form>
</body>
```

Labels this part of the code in case we want to refer to it later

Tells the user what the webpage does (`h2` makes it a headline font)

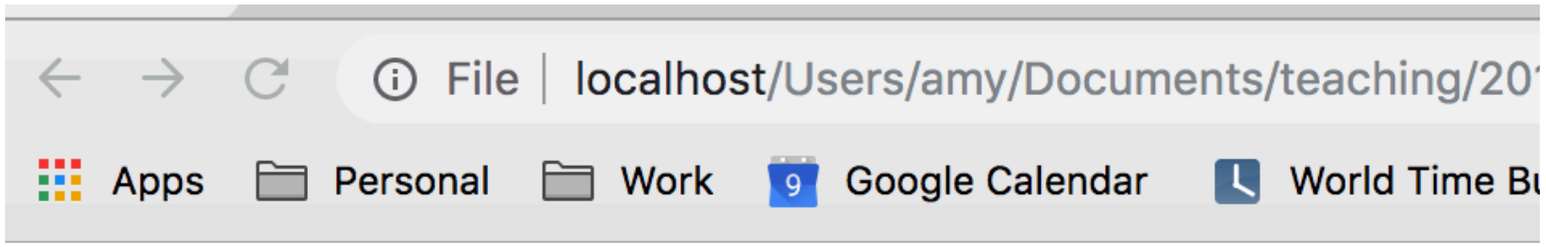
Creates an input box which is initialised at value 0, and calls it `DegC` (for the function to refer to)

Creates a button which when clicked calls the `temperature()` function we just created, and sends it this form element

Once the script calculates `DegF` this is made visible in this readonly element

HOW DOES JAVASCRIPT COME IN?

Give it a try!



Celsius to Fahrenheit Converter

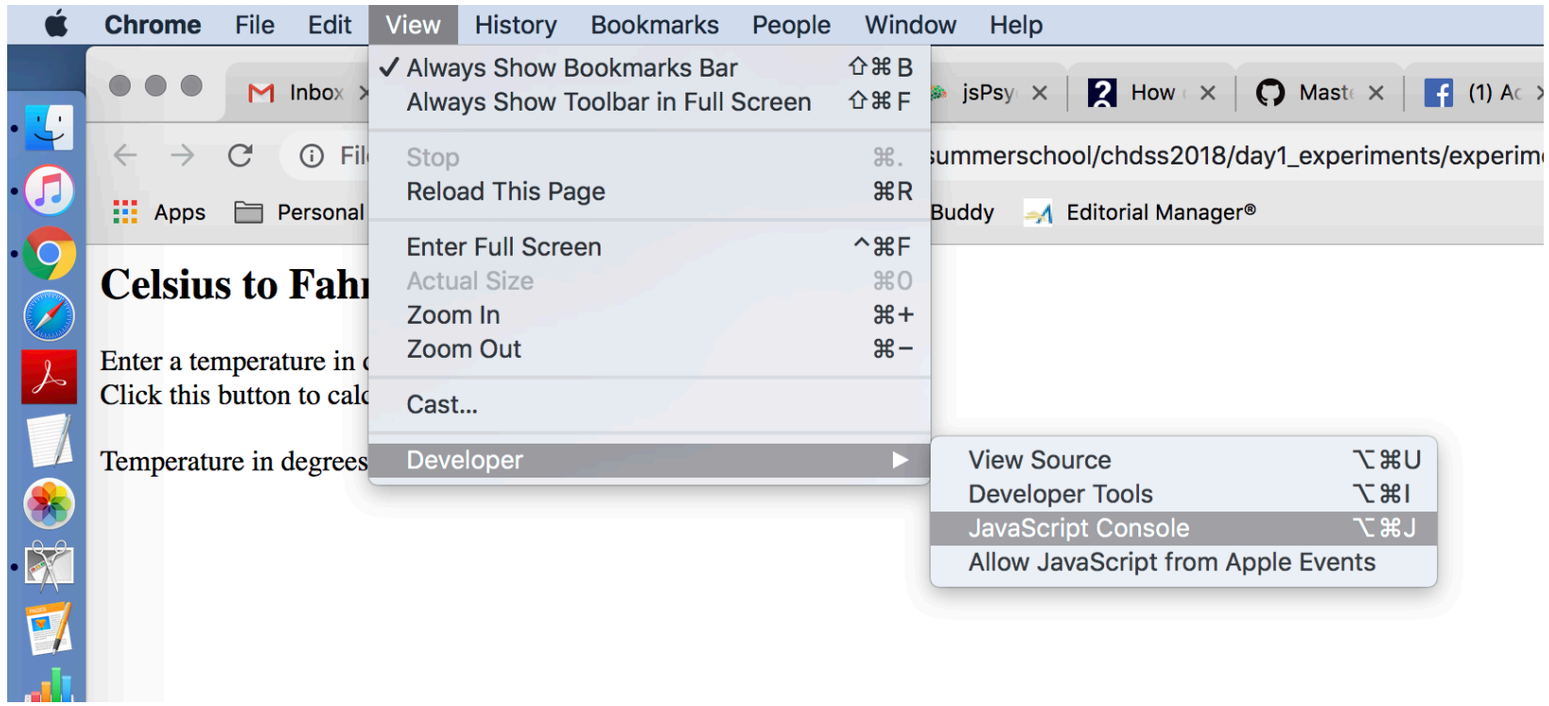
Enter a temperature in degrees C: 34
Click this button to calculate the temperature in degrees F: Calculate

Temperature in degrees F is: 93.2

HOW DOES JAVASCRIPT COME IN?

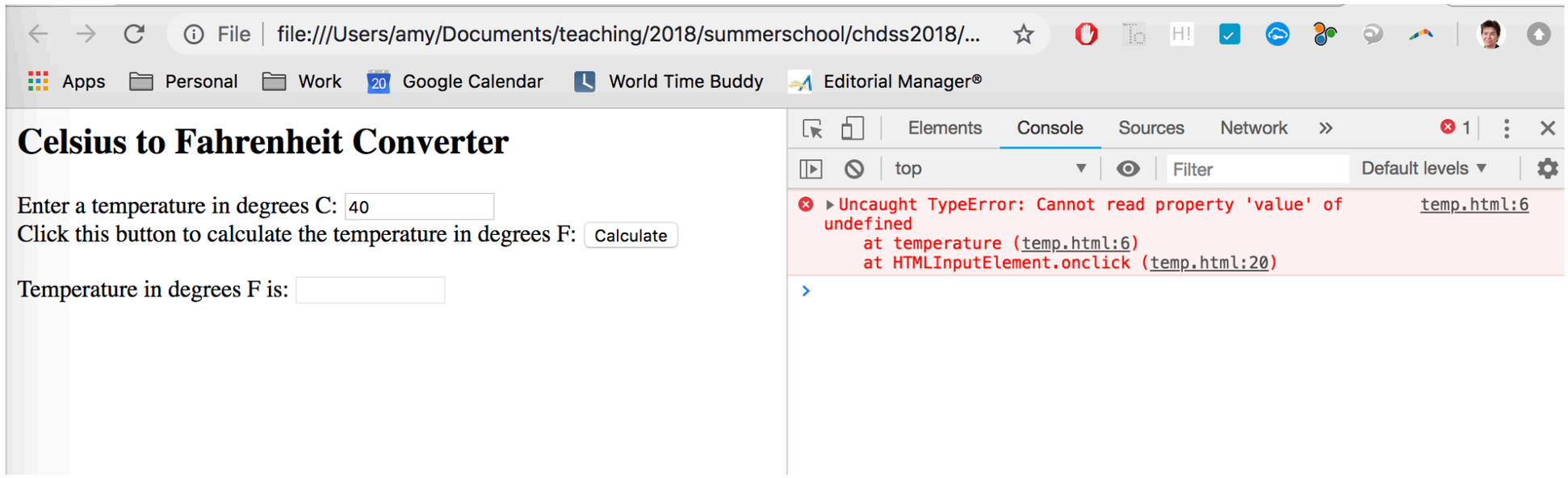
What if you screwed up?
In Chrome, go to Javascript console, which gives you error messages

```
<script language="JavaScript">
  function temperature(form) {
    var c = parseFloat(form.DeggC.value, 10);
    var f = 0;
    f = c * (9.0/5.0) + 32;
    form.DegF.value = f;
  }
</script>
```



HOW DOES JAVASCRIPT COME IN?

What if you screwed up?
In Chrome, go to Javascript console, which gives you error messages



EXERCISE

Try modifying this function to convert from Fahrenheit to Celcius instead

A SUPER USEFUL TOOL: JSPSYCH

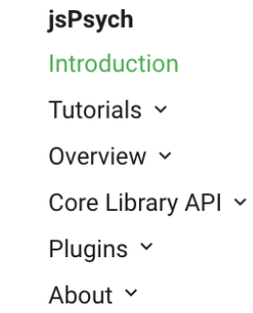
jsPsych is a library of javascript functions designed specifically to administer web experiments

We do not have time to make you experts in this, but they have great tutorials on their webpage:

https://www.jspsych.org/

A SUPER USEFUL TOOL: JSPSYCH

The way jsPsych works is by creating a bunch of plugins that are Javascript code you can call to do some of the complicated stuff in your experiment



You can download jsPsych and the plugins from the webpage if you want to work with it directly (again, we won't be)

A SUPER USEFUL TOOL: JSPSYCH

To include jsPsych and the plugins in your html file, you can just link to them rather than writing out the whole script!

Imports the main jsPsych library

```
<html>
  <head>
    <title>Sample Experiment</title>
    <script src="./js/jspsych.js"></script>
    <link href="./js/css/jspsych.css" rel="stylesheet" type="text/css"></link>
  </head>
  <body>
  </body>
</html>
```

Imports a stylesheet, which basically is a set of guidelines making the visual presentation nice

A SUPER USEFUL TOOL: JSPSYCH

Then you can include code in the body which gives those plugins the info they need to work

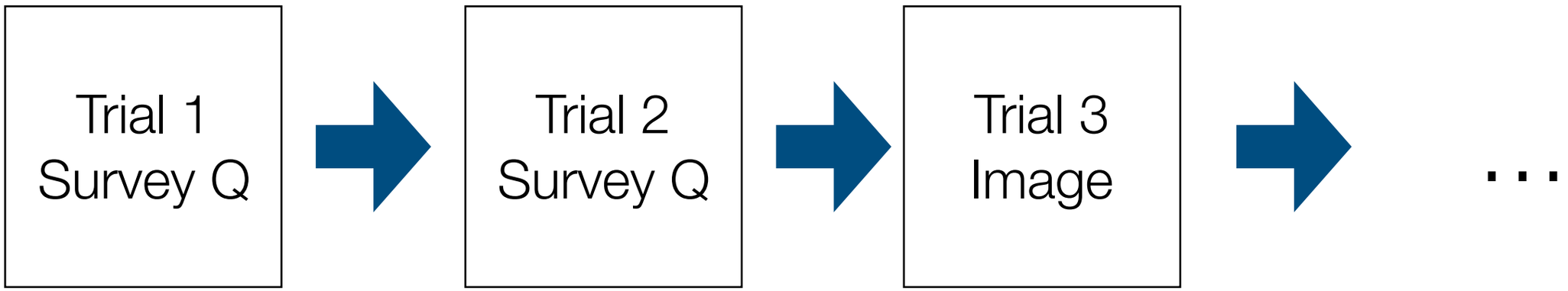
```
<body>
<!-- Starting screen -->
<div class="start" style="width: 1000px">
  <!-- Text box for the splash page -->
  <div class="start" style="text-align:left; border:0px solid; padding:10px;  width:800px; floa
    <p><b>"The Spheres of Sodor"</b> is a short psychological study investigating how people ma
    <!-- Next button for the splash page -->
    <p align="center">
    <input type="button" id="splashButton" class="start jspsych-btn" value="Start!"
                        onclick="splashButtonClick()"> </p>
    </div>
  </div>

  </body>

<script>

  // Some basic functions
  function splashButtonClick() {
    setDisplay('start', 'none');
    setDisplay('consent', '');
  }

  // Function to change the display property of a set of objects
    function setDisplay(theClass, theValue) {
      var i, classElements = document.getElementsByClassName(theClass);
      for (i = 0; i < classElements.length; i = i + 1) {
          classElements[i].style.display = theValue;
      }
    }

</script>
```

A SUPER USEFUL TOOL: JSPSYCH

Jaysire (the R package we'll be learning) handles all of this. I just wanted to give you a sense of what it's doing when we use it - it outputs experiment code in javascript which does that

TIMELINES

The way jsPsych and jaysire both work is by building a description of an experiment known as a **`timeline`**, which is basically a series of variables defining each step (trial).



Timeline

You can create and randomise variables, nest timelines within one another, present audio / visual / text / images, create lots of different kinds of questions, etc.

JAYSIRE: STRUCTURE

Collections of functions that do different things:
https://djnavarro.github.io/jaysire/reference/

`build_`	Builds the experiment in javascript
`trial_`	Functions for creating lots of different kinds of trials
`tl_`	Functions for putting trials together into a timeline
`run_`	Functions for running the experiment
`fn_`	Manipulates javascript functions (advanced)

JAYSIRE: INSTALLATION

```
remotes::install_github("djnavarro/jaysire")
```

loads it onto your machine

```
> remotes::install_github("djnavarro/jaysire")
Downloading GitHub repo djnavarro/jaysire@master
These packages have more recent versions available.
Which would you like to update?

1: All
2: CRAN packages only
3: None
4: rlang  (0.4.1  -> 0.4.2 ) [CRAN]
5: digest (0.6.22 -> 0.6.23) [CRAN]

Enter one or more numbers, or an empty line to skip updates:
1
rlang  (0.4.1  -> 0.4.2 ) [CRAN]
digest (0.6.22 -> 0.6.23) [CRAN]
Installing 2 packages: rlang, digest

  There are binary versions available but the source versions are later:
      binary source needs_compilation
rlang   0.3.1  0.4.2              TRUE
digest 0.6.18 0.6.23              TRUE

Do you want to install from sources the packages which need compilation?
y/n: y
```

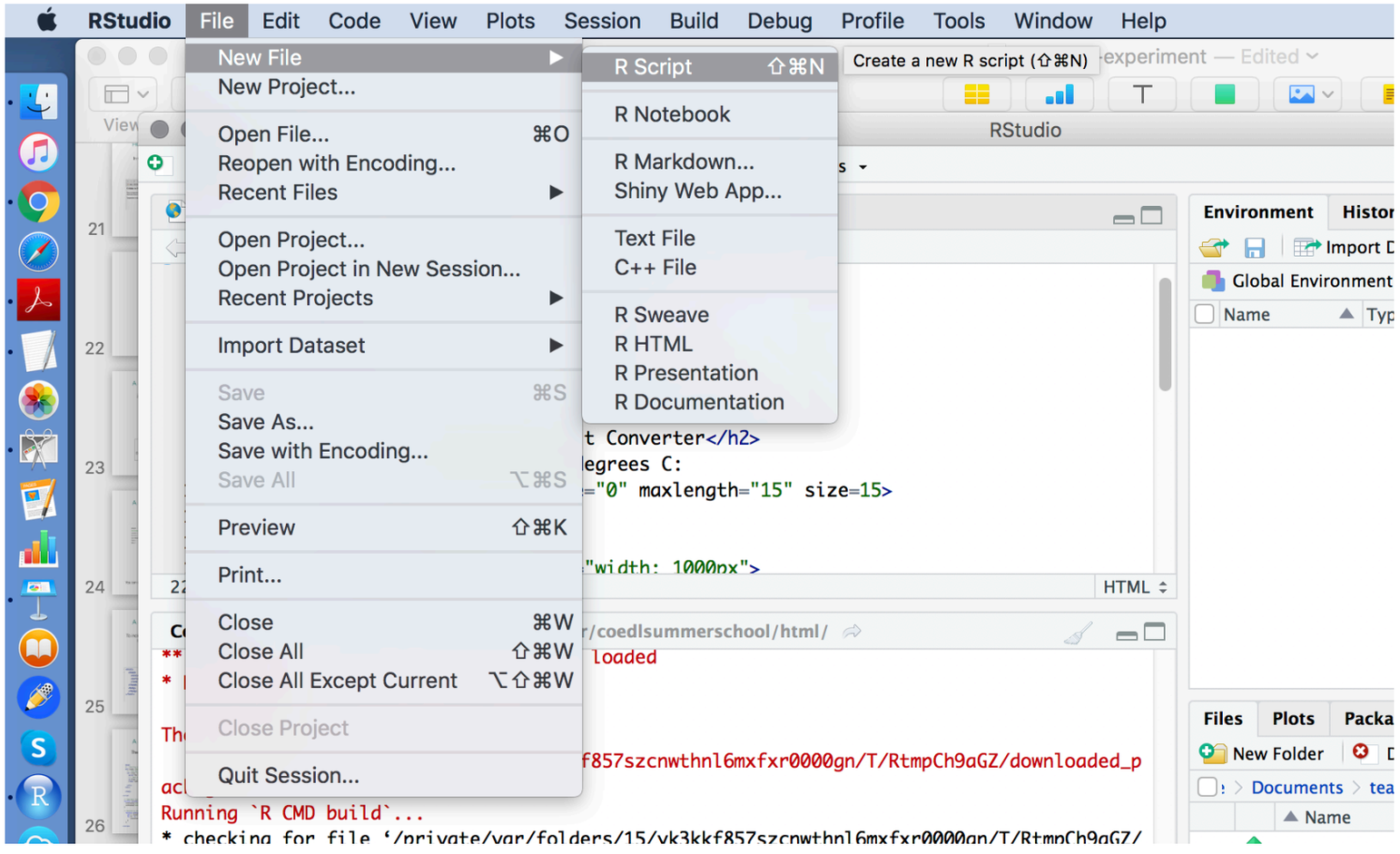
puts it in working memory (need this in every experiment you make)

```
library(jaysire)
```

JAYSIRE: GETTING STARTED

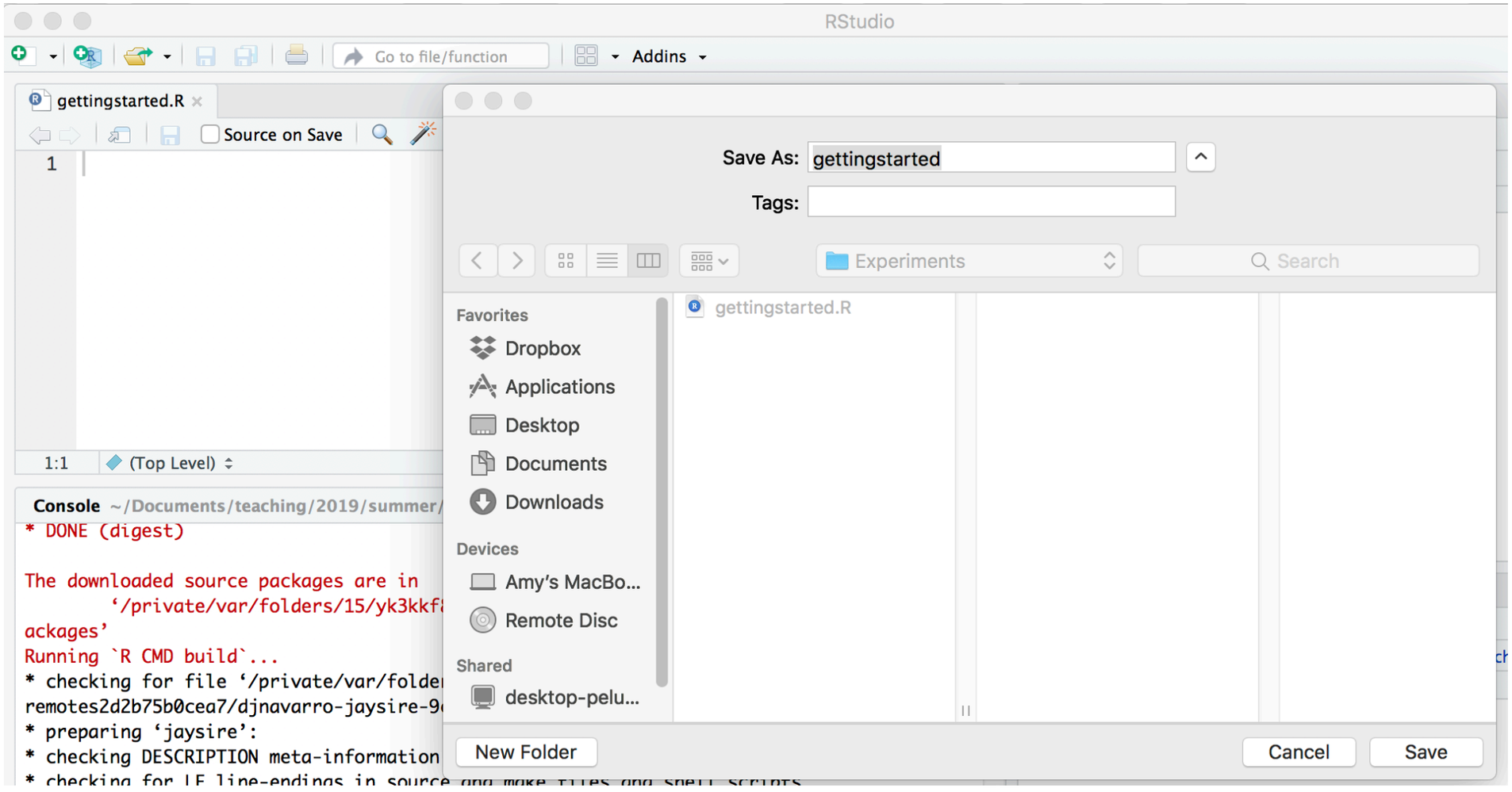
Let's work through the first online tutorial together:
https://djnavarro.github.io/jaysire/articles/jaysire01.html

First, create an empty R script for our experiment.



JAYSIRE: GETTING STARTED

Make a new folder called “Experiments”, go to it, and save the R script there as **`gettingstarted.R`**



JAYSIRE: GETTING STARTED

Let's start by creating instructions using the function called `trial_instructions()`

```
help(trial_instructions)
```

Takes a bunch of arguments which describe the instructions people will see

R: Specify pages of instructions to display — Find in Topic

Specify pages of instructions to display

Description

The `trial_instructions` function is used to display one or more pages of instructions that a participant can browse.

Usage

```
trial_instructions(pages, key_forward = "rightarrow",
  key_backward = "leftarrow", allow_backward = TRUE,
  allow_keys = TRUE, show_clickable_nav = FALSE,
  button_label_previous = "Previous", button_label_next = "Next",
  post_trial_gap = 0, on_finish = NULL, on_load = NULL,
  data = NULL)
```

Arguments

`pages`	Character vector. Each element should be an HTML-formatted string specifying a page
`key_forward`	This is the key that the subject can press in order to advance to the next page, specified as their numeric key code or as characters
`key_backward`	This is the key that the subject can press in order to return to the previous page.
`allow_backward`	If TRUE, participants can navigate backwards
`allow_keys`	If TRUE, participants can use keyboard keys to navigate
`show_clickable_nav`	If TRUE, buttons will be shown to allow navigation
`button_label_previous`	Text on the "previous" button
`button_label_next`	Text on the "next" button
`post_trial_gap`	The gap in milliseconds between the current trial and the next trial. If NULL,

JAYSIRE: GETTING STARTED

Let's start by creating instructions using the function called `trial_instructions()`

Putting the instructions into your own variable called instructions

The function name (needs parentheses around arguments)

These are the three arguments to the function:

```
instructions <- trial_instructions(
  pages = c(
    "Welcome! Use the arrow buttons to browse these instructions",
    "Press the 'Next' button to begin!"
  ),
  show_clickable_nav = TRUE,
  post_trial_gap = 1000
)
```

The two pages are in a vector (the command `c()` does this) separated by commas

How long to wait after clicking before going to the next page

Let participants click back and forth

JAYSIRE: GETTING STARTED

Now let's build a minimal experiment with just these instructions

The function name (needs parentheses around arguments)

Builds a timeline consisting only of the instructions we created

```
build_experiment(
  timeline = build_timeline(instructions),
  path = file.path("starting_exp"),
  on_finish = fn_save_locally()
)
```

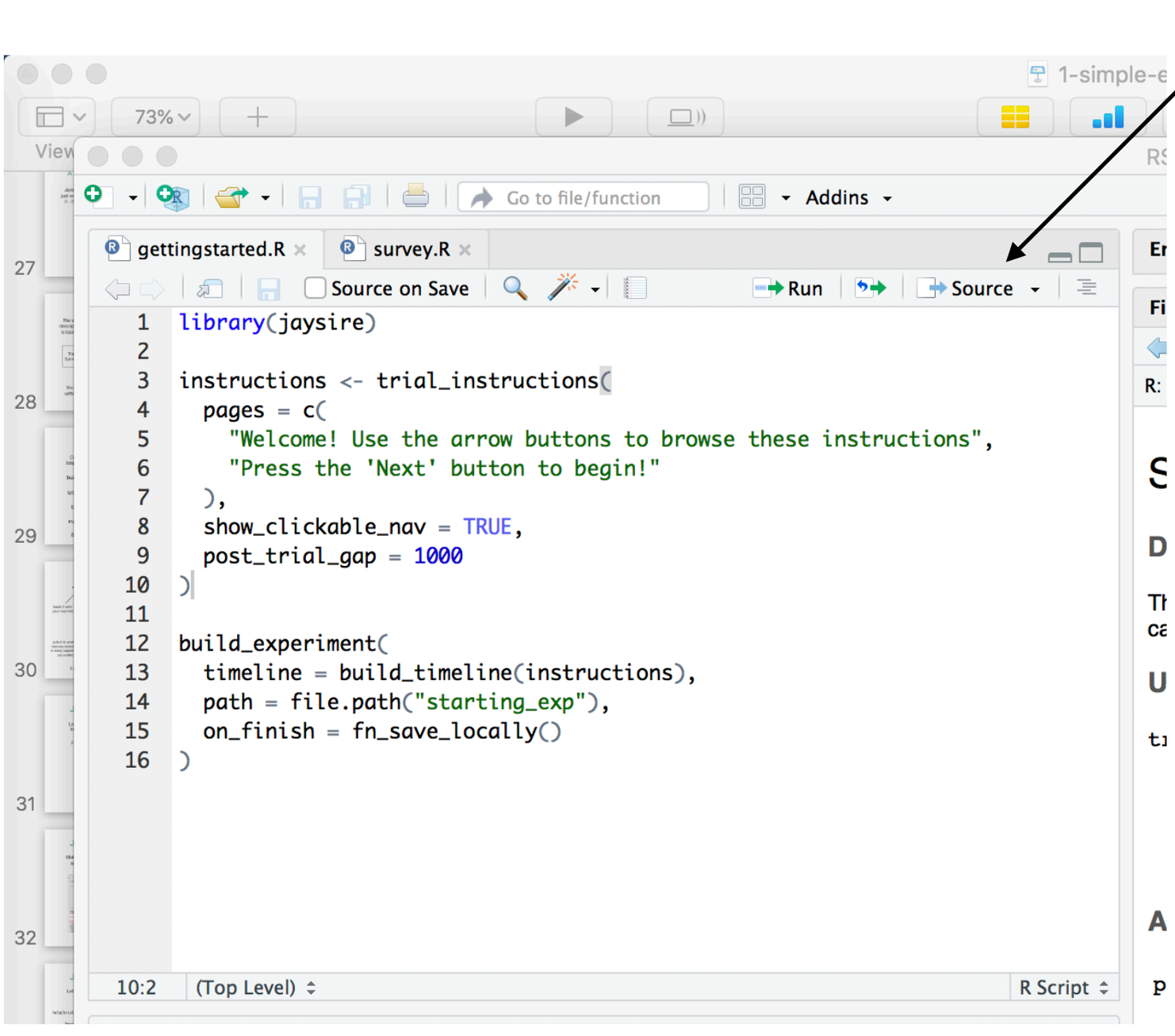
These are the three arguments to the function:

Tells R to save the data locally on your computer

Tells R to put the experiment in a new[*] folder called `starting_exp` in your current directory

JAYSIRE: GETTING STARTED

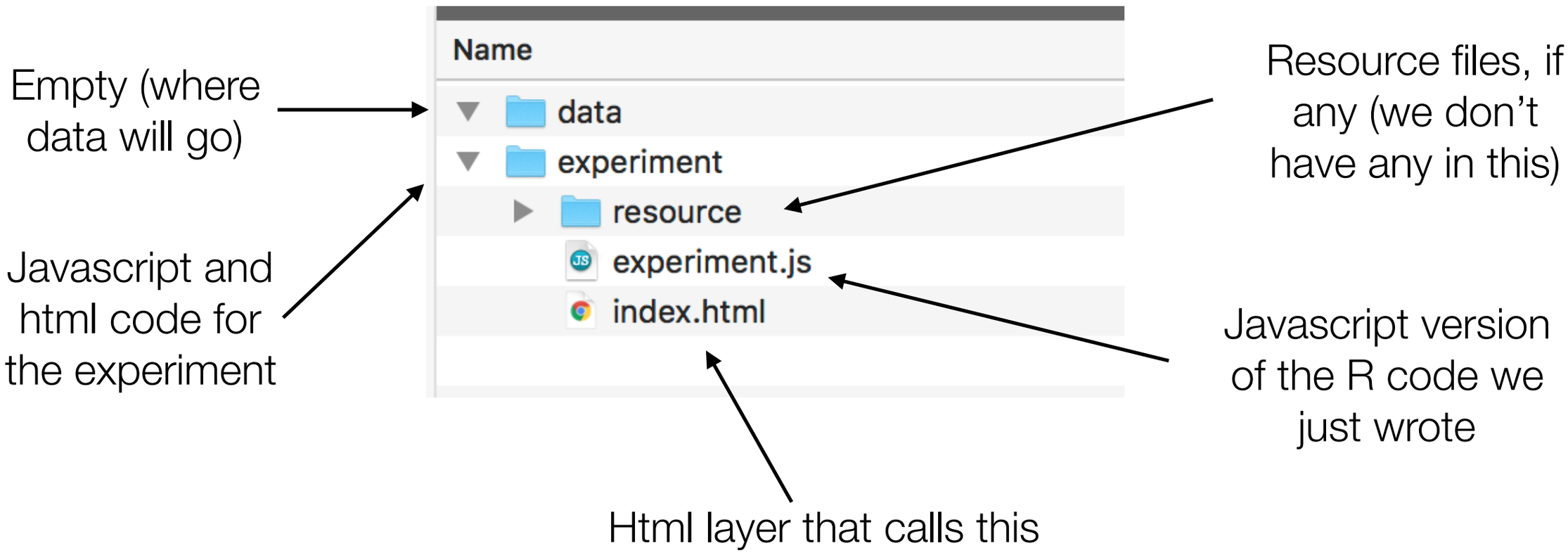
"Source" the experiment in R to get it ready to run



JAYSIRE: GETTING STARTED

"Source" the experiment in R to get it ready to run

You'll see that this builds the experiment - creates two folders inside `starting_exp`



JAYSIRE: GETTING STARTED

Run it by using the function `run_locally()`.
Type it at the console with your path in as an argument

The function name (needs parentheses around arguments)

Tells R this is a path

Name of the folder with the experiment

```
> run_locally(file.path("starting_exp"))
Starting server to listen on port 8000
```

EXERCISE

Try sourcing it again: what happens? Why?

It assumes that `starting_exp` folder doesn't contain any subfolders. Since it does (you've already built it) it throws an error. Can avoid this by deleting the `data` and `experiment` folders each time you source again (tedious) or add in this bit to your code:

```
# set directory (deletes any existing old experiment builds in it)
my_directory <- file.path("classsurvey_exp")
# create the empty folder if necessary
if(dir.exists(my_directory)) {
  unlink(my_directory, recursive = TRUE)
}
```

And then replace the argument in build_experiment:

```
path = my_directory,
```

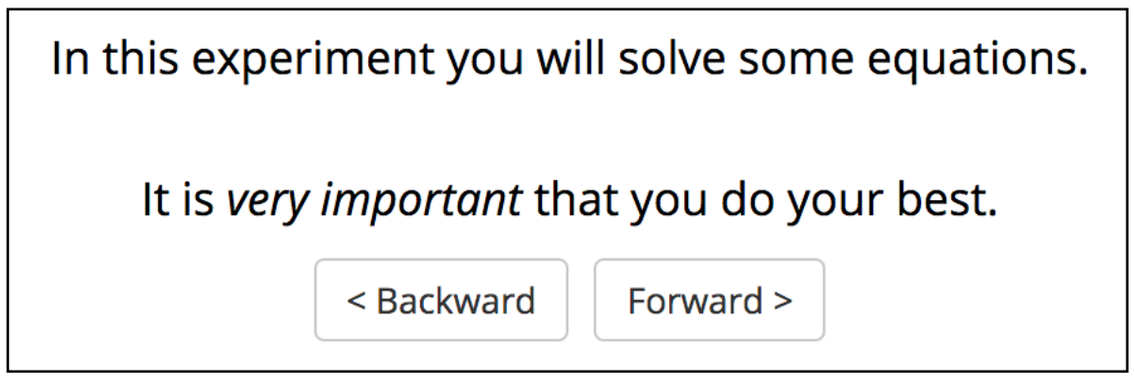
How do you run it this time?

EXERCISE

Change your instructions so they look like this:

Page 1

Page 2



Page 3

Hints: remember to use `help(trial_instructions)` to figure out how to change the button labels. And the text in the instruction pages can be formatted using html tags.

ADDING A TRIAL

Let's make a trial that presents the stimulus as html and collects a response using buttons:

Putting this trial into your own variable called trial1

The function name (needs parentheses around arguments)

The stimulus shown to the participant

The button choices and labels

How long to wait after clicking before going to the next page

```
trial1 <- trial_html_button_response(
  stimulus = "13 + 23 = 36",
  choices = c("true", "false"),
  post_trial_gap = 500
)
```

ADDING A TRIAL

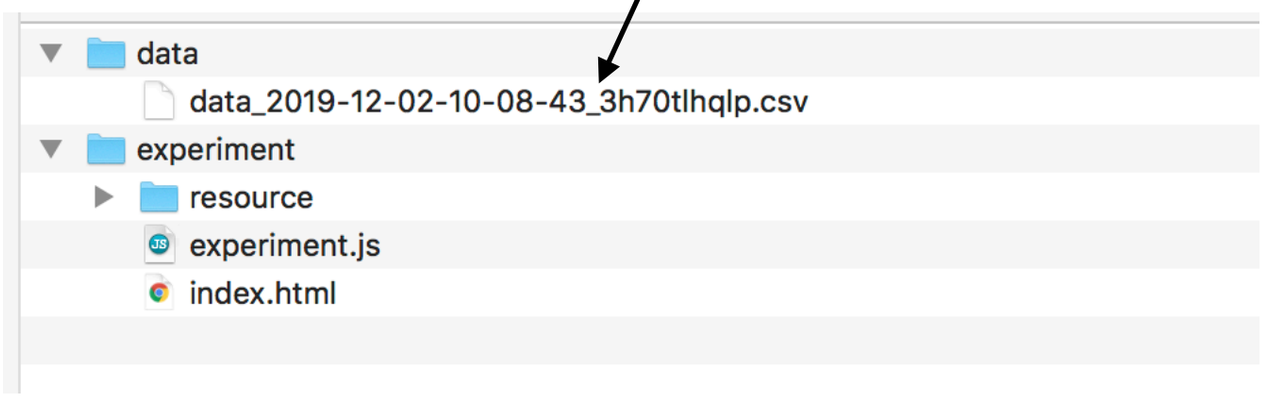
Let’s make a trial that presents the stimulus as html and collects a response using buttons:

Need to also remember to add it to the timeline!

```
build_experiment(
  timeline = build_timeline(instructions,trial1),
  path = my_directory,
  on_finish = fn_save_locally()
)
```

ADDING A TRIAL

This time when we run it there is data to save, which goes in the `data` folder



It’s saved as a csv in a format called JSON format, which I’ll give you a script to turn into manageable form. Still even now you can see what is there

EXERCISE

Add another trial which shows the equation `2+2=5` for only 500ms before it disappears.

The participant should have three response options: `true`, `false`, and `I don't know`