REACTION TIME DEMOS

Psychophysics: Theory and Application | 30 Oct. – 2 Nov., 2023

Contents

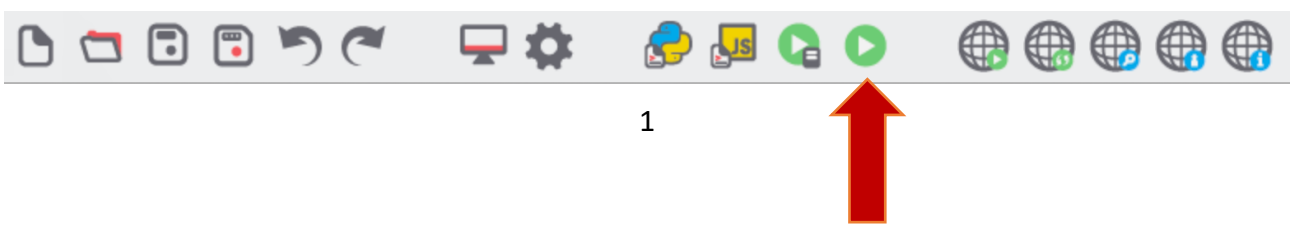
Additionally, if you find you've moved faster than the pace of the class, you can try your hand at the bonus **challenge tasks.**

Part 1: Example Experiment

Description: To familiarize yourself with what it's like to participate in a reaction time experiment—as well as to verify that PsychoPy is downloaded and functioning properly—you'll start out by completing a demo experiment similar to one discussed in the RT lecture last week. Your instructions for this experiment: press SPACEBAR as soon as you see any shape.

Procedure:

1. Download the *RT_Demos* folder from the *2023 - Psychophysics Material* Seafile directory. (https://seafile.utu.fi/d/00eb16a8e5fc409e90eb/)
2. Locate the PsychoPy program in your desktop and open it.
3. From File/Open, select *example_experiment.psyexp* in the *example_experiment* folder. (Note: You must download the full contents of the *example_experiment* folder for the experiment to run, and folder contents must be in their original configurations.
4. Once the experiment file has loaded, you can begin the experiment by clicking on the "run experiment" icon in the menu bar (below). You'll be instructed on what to enter for "participant."



5. Once you have completed the experiment, upload this data to the *group_data* folder in the *RT_demos* Seafile (link: https://seafile.utu.fi/u/d/cc084cd7aec9434dbc6e/). We'll use this data later on for the group-level analyses. The data file will be located in the *data* folder within *example_experiment,* and it will have the format subjectName_example_ experiment_timeCode.csv

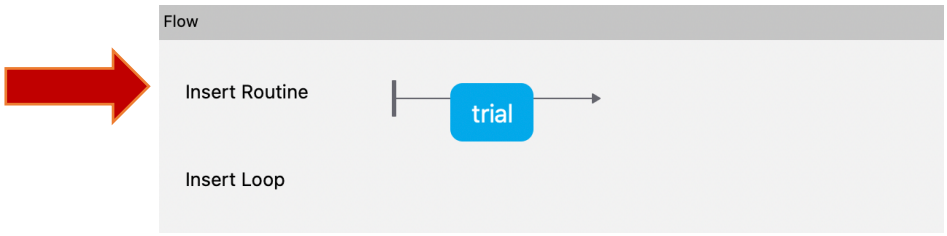
Part 2: PsychoPy Builder (Feature Walkthrough)

Description: Builder is the graphical user interface for PsychoPy. Here we'll walk through some of the core elements of builder—routines, stimuli and response components, and loops—to build a very basic experimental architecture. For more information on the many other features of Builder, visit https://psychopy.org/builder/index.html.

1. **Routines:** Routines are the basic building blocks of PsychoPy Builder, the fundamental unit into which all experiments are organized. Every event in an experiment, including stimulus presentation and response registration, is executed within a routine. In the flow of the experiment, routines are ordered sequentially. Routines have a definite temporal structure: Within each routine, the onset and duration of stimulus and response components can be tightly controlled, and multiple components can appear in parallel.

   Design Tip: Try to minimize the number of routines necessary to realize your experiment. Many designs require no more than 1 or 2 routines.

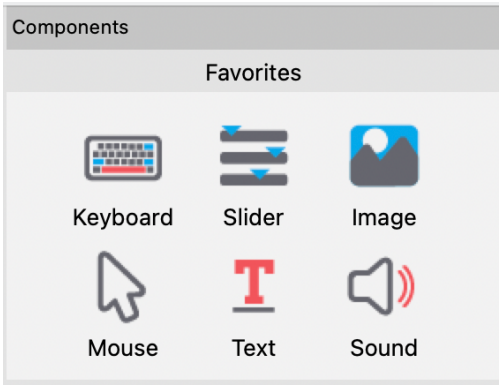
   A new file in Builder will automatically have a routine named "trial." New routines can be added by selecting "insert routine" at the bottom of the screen.

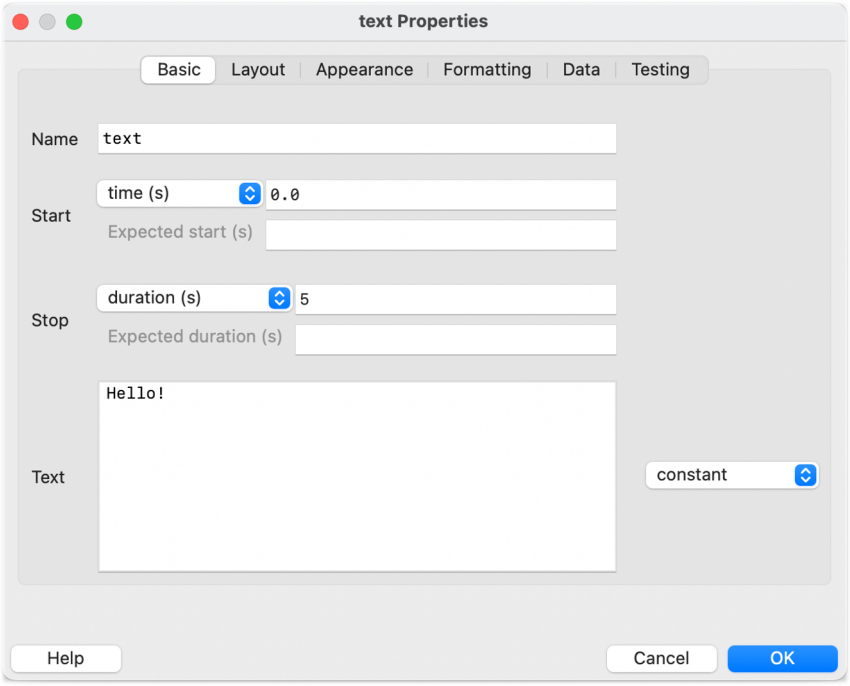


2. **Text Component:** As the name suggests, this is the component for presenting text stimuli. To add this component, select the "text" icon from the "components" menu at the top right of the screen.

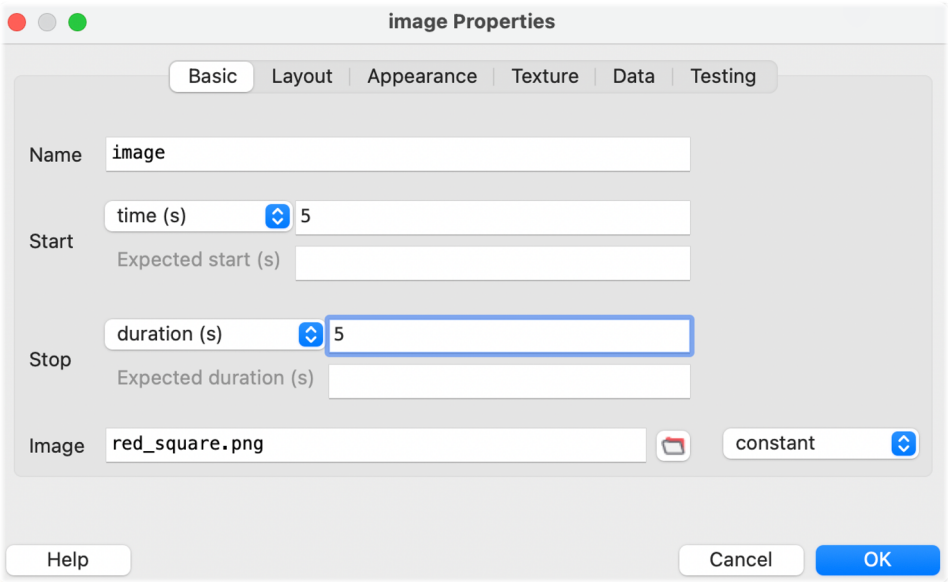
   Once you have selected the text component, you will be given a menu that specifies a number of component parameters, including onset, duration, and content of the text.

   Important: If you leave the duration of a component blank, it will continue indefinitely. This means the routine won't end until a response component forces it to (or the participant quits the experiment).

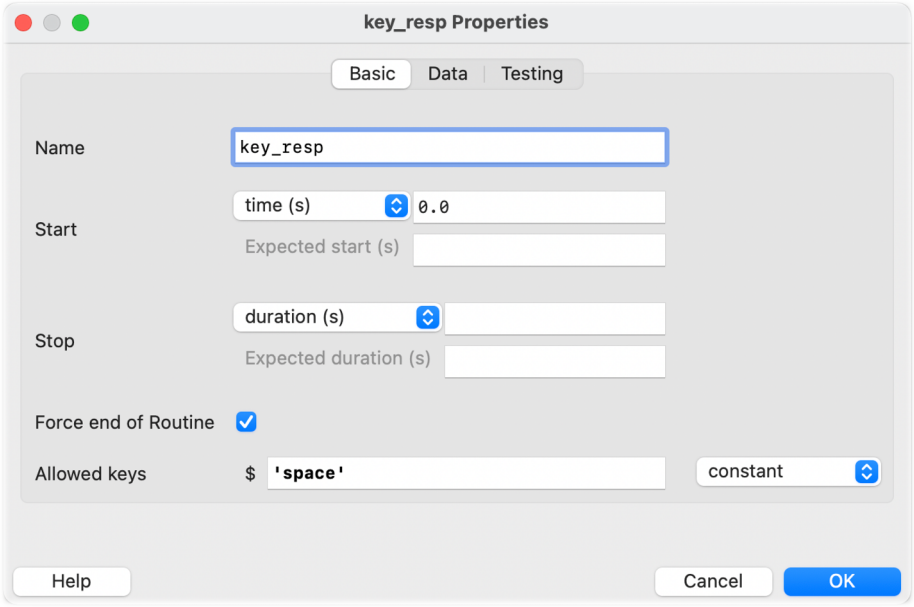




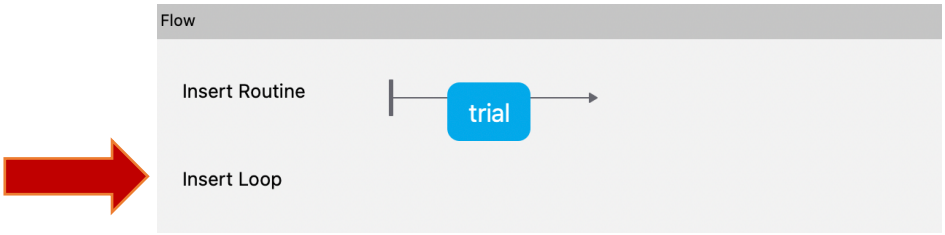
3. **Image Component:** The image component is structurally similar to the text component, but now it presents a specified image file as a visual stimulus. If the image file and PsychoPy experiment are stored in different locations, the image path should be specified from the location of the experiment. Note that sometimes the default size and position settings might not be ideal for the particular image, and those can be changed in the "layout" tab.



4. **Keyboard Component:** This is the main response component we'll use in this course. It allows you to register specific responses from participants, as well as force-end routines once a response is registered.



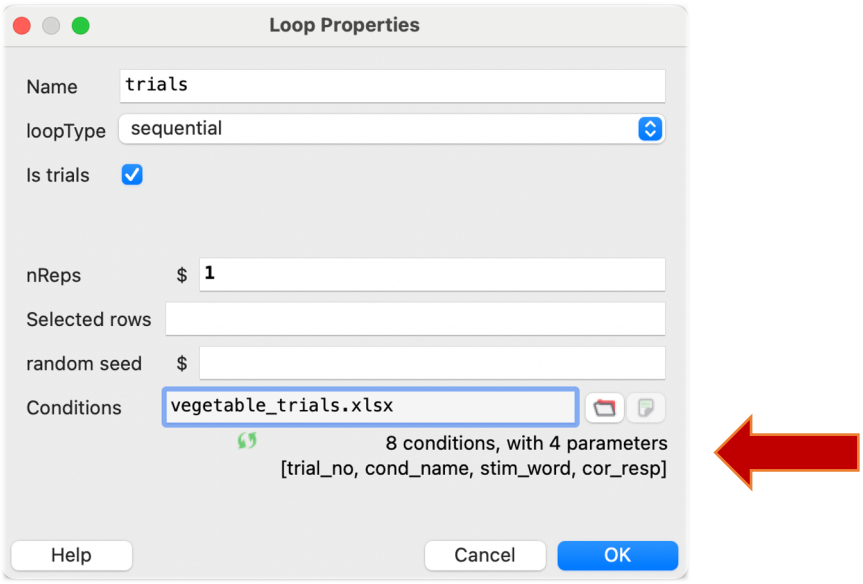
5. **Loops:** Loops allow you to repeat routines (and sequences of routines.) When inserting a loop, you can select which routine(s) you want it to loop over.



Loops are especially important in PsychoPy, because they allow you to specify the values of variables for the routines that they loop over. This is done with spreadsheets. For example, say you have a go/no-go task in which participants are supposed to respond for the word "cabbage" and withhold response for all other words. The 8 trials for this experiment are organized into a simple spreadsheet, which we'll call *vegetable_trials.xlsx*:

trial_no	cond_name	stim_word	cor_resp
1	Cabbage	Cabbage	1
2	Cabbage	Cabbage	1
3	Cabbage	Cabbage	1
4	Cabbage	Cabbage	1
5	Non-cabbage	Beet	0
6	Non-cabbage	Turnip	0
7	Non-cabbage	Carrot	0
8	Non-cabbage	Radish	0

We can then tell PsychoPy to loop over each of these conditions by specifying this spreadsheet in the loop properties:



PsychoPy will automatically take items in the first row to be "parameters," or variable names you can reference in routines within the loop. Subsequent rows are handled as "conditions" (a bit of a misnomer). These are the values of your variables for a given loop.

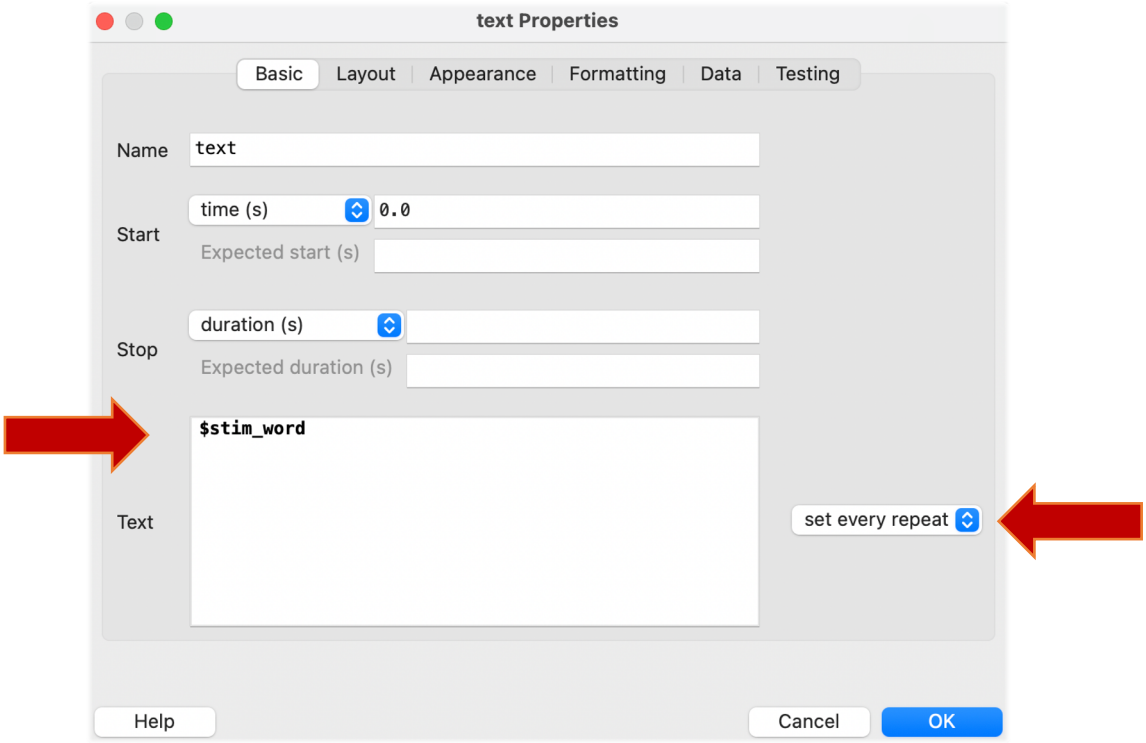
Loops iterate through each row one-by-one. If the loop type is specified as "sequential," it will proceed through each row in order. You can also specify random or staircase loops. Additionally, you can also specify that a loop proceed through your spreadsheet multiple times ("nReps") or for only some designated rows ("Selected rows"). When selecting rows, remember that PsychoPy uses Python-style row numbering, **not** the numbering from the Excel sheet: row 0 in PsychoPy will select what Excel calls row 2 (the first trial in the spreadsheet), 0:3 in PsychoPy will select the first 3 trials (rows 0, 1, and 2 using Python-style indexing; which are rows 2, 3, and 4 in Excel) and so on.

Design Tip: In addition to .xlsx files, you can also use .csv files for loops.

6. **Variables in PsychoPy Builder:** Loops are the most straightforward way to introduce variables in Builder. In the example above, we introduced 4 variable names—*trial_no, cond_name, stim_word, cor_resp*—along with the values those variables can assume.

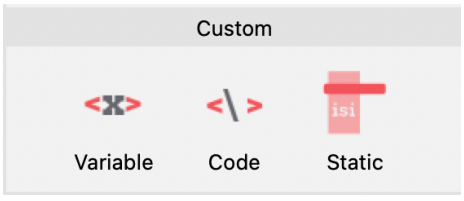
   To reference these variables in specific components, PsychoPy uses the $ + variable name format. This tells PsychoPy to interpret the input as a line of Python. For example, to use the variable *stim_word* in a text component, we can set the text to be $stim_word. Now the presented text will correspond with the values specified in the "stim_word" column of the spreadsheet.

Important: When using variable names in the field of a stimulus or response component, it's critical to **set the field to update every repeat.** *Otherwise, it won't work.*



Design Tip: Assigning variable names through loops allows you to easily control the presentation of different images for different trials or conditions. Set a column header as your variable name (e.g., imageFile) and the subsequent rows of the column as the files (e.g., file1.jpg, file2.jpg, etc.). You can then use $imageFile in an image component to access all those files.

In addition to using loops, you can also assign variables through the code component—which allows you to directly insert Python (or JavaScript) code directly into a routine—or the designated variable component. Using the code component allows you to write scripts for more complex operations on variables (e.g., providing feedback).



Since the experiment script compiles in the order that components are listed, make sure that code components are positioned before any stimulus or response components that use variables introduced in code components. (You can also introduce variables in the "before experiment" tab, passing default values to make debugging easier.)

Part 3: PsychoPy Builder Practice

Description: Now we'll practice by building a simple experiment to investigate the effect of *foreperiod*, the length of time between a warning signal and the target stimulus, on RT.

This experiment should have the following characteristics:

1. It should use a **2-choice design,** in which the task is to identify whether a presented visual stimulus is a green square or a red square. Images of green and red squares are available in *RT_Demos/stimuli_squares* (green_square.png and red_square.png).

2. Each square should appear for a maximum of **500 ms.**

3. The **response component** should begin concurrently with the presentation of the square. Participants should have an additional 1 second after the square disappears to respond, after which the routine should time out. Participants should respond with the right key for red and the left key for green.

4. There should be a **1 second warning signal** at the beginning of each trial.

5. The experiment should have **two conditions,** a long onset condition and a short onset condition. For the short onset condition, the target stimulus should be presented 500 ms after the warning signal. For the long onset condition, it should be 2 s.

6. The experiment should use a **block design:** It should first present all the trials in the short condition before then presenting all the trials in the long condition.

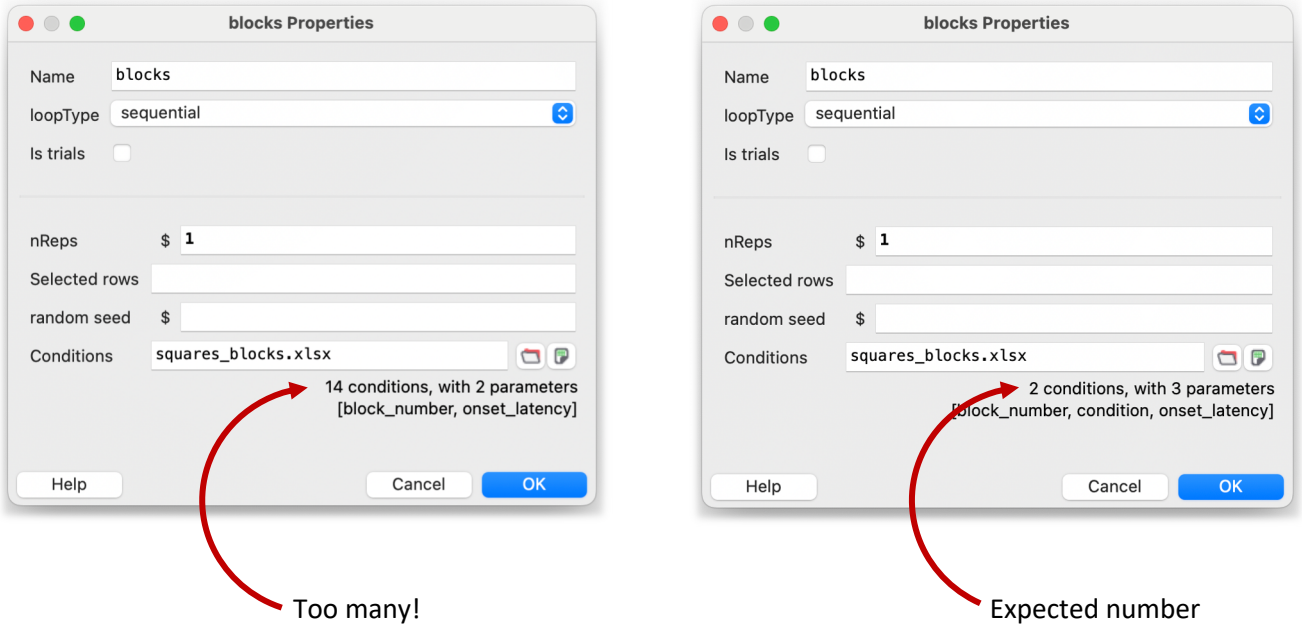
7. There should be **20 trials per condition.** Each condition should have an equal distribution of green and red stimuli, and their order should be random within each block.

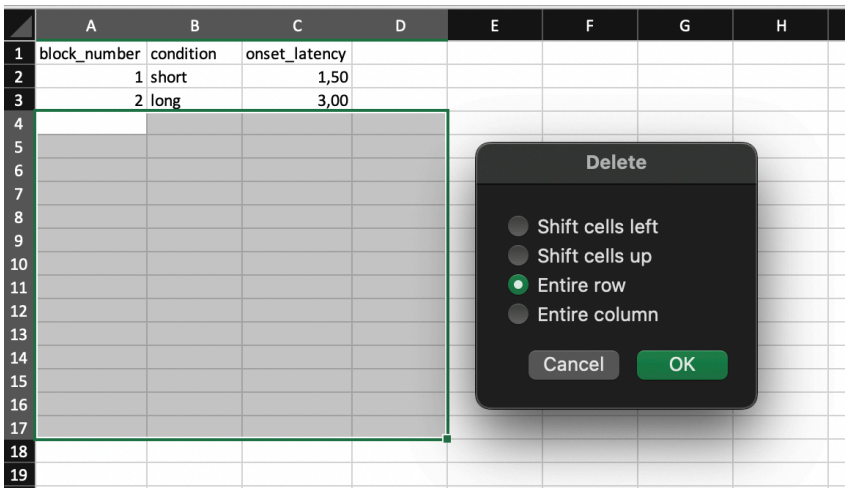
8. You should be able to build this experiment using only **one routine** (comprised of **three components**), and at most **two loops and spreadsheets**. It is also possible to build with only one loop, but this will make randomization within blocks more difficult.

Once you have built the experiment, you can try it yourself. Remember to complete it as quickly as possible while still being accurate.

> *Important: When naming your variables in Excel sheets, be careful not to use names that are already in use by default in PsychoPy (e.g., "trial" or "text"). If you're not sure, try re-naming a routine to that name. PsychoPy will tell you if it's unavailable.*

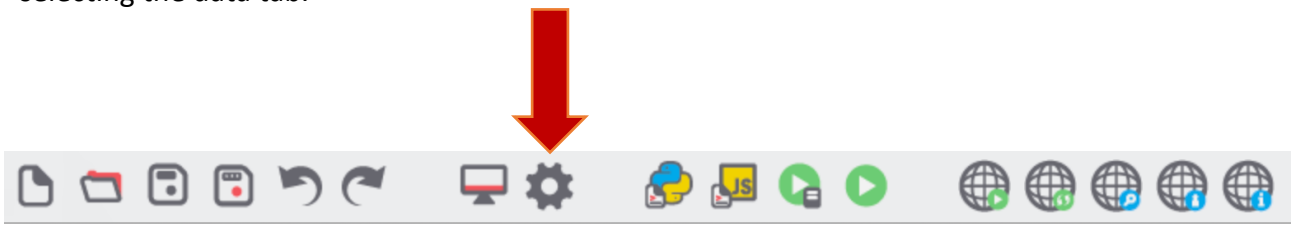
Finally, watch out for PsychoPy registering empty cells in Excel spreadsheets as part of your data input. Check to make sure that the number of conditions matches what you expect. If it's too many, your experiment won't end like it should. Go back to the spreadsheet and "delete" the empty rows.





Part 4: Understanding the Output Data File

PsychoPy provides a number of options for the output of data files. In Builder, these options can be accessed in the "data" tab of the settings menu. Here we will be using the default output .csv file. You can access data output settings by going to the settings menu and selecting the data tab.

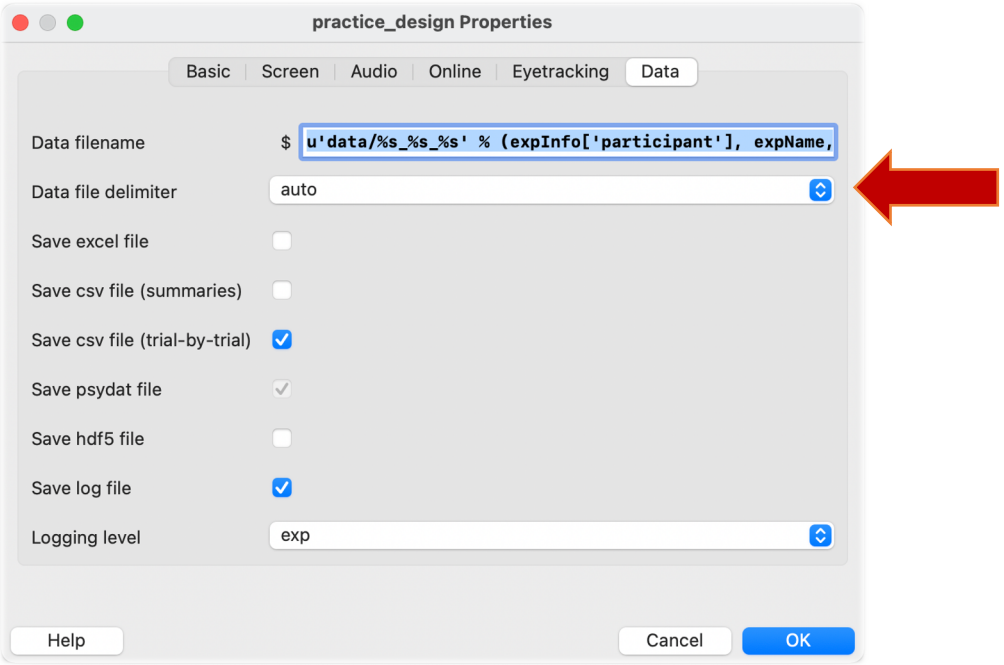


By default, output data files are stored in a “data” folder located in the same directory as the experiment. (You can change the output destination and naming conventions in the data settings.) **Open this file in a text editor** (e.g., VS Code), and you’ll see its raw structure—columns of data separated by a “delimiter.” In this example the delimiter is a comma, which is generally the default.

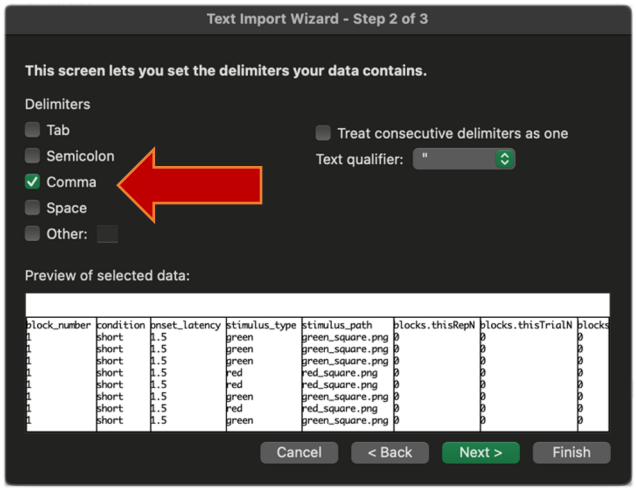
demo_data.csv

```
block_number,condition,onset_latency,stimulus_type,stimulus_path,blocks.thisRepN,blocks.thisTria
1,short,1.5,green,green_square.png,0,0,0,0,0,0,0,0,11.893016330999671,12.906596414000887,13.3733
1,short,1.5,green,green_square.png,0,0,0,0,0,1,1,5,13.788233528999626,14.790810479000356,15.2741
1,short,1.5,green,green_square.png,0,0,0,0,0,2,2,8,15.645088925000891,16.657712714999434,17.1411
1,short,1.5,red,red_square.png,0,0,0,0,0,3,3,15,17.478697839000233,18.492139772000883,18.9755055
1,short,1.5,red,red_square.png,0,0,0,0,0,4,4,18,19.295153244000176,20.308836186000917,20.7923546
1,short,1.5,green,green_square.png,0,0,0,0,0,5,5,6,21.180093009999837,22.19245628799945,22.67600
1,short,1.5,red,red_square.png,0,0,0,0,0,6,6,14,23.030163089999405,24.043481873999553,24.5269016
1,short,1.5,green,green_square.png,0,0,0,0,0,7,7,9,24.929140312000527,25.94369261599968,26.42709
1,short,1.5,red,red_square.png,0,0,0,0,0,8,8,17,26.73073251599999,27.744040330000644,28.22735237
1,short,1.5,red,red_square.png,0,0,0,0,0,9,9,19,28.565637024999887,29.578297510000993,30.0615992
```

Knowing the delimiter of your data file will be crucial for successfully reading and processing that data. If you want to change the delimiter of the files output by Builder, you can do this too in the data tab of the settings menu.



To open a .csv file in Excel, you may have to specify the delimiter for that file. This can be done by selecting File/Open in Excel (rather than opening the file directly from the folder). You can then preview the imported file and select the correct delimiter.



Observe that the data file is organized with one data type per column, with the first row identifying the data type of each column. Each row corresponds with one trial, with rows of data recorded sequentially. The first columns correspond with the names of the variables passed to the loops. In this example, block_number, condition, and onset_latency all come from the spreadsheet used for the block loop, and stimulus_type and stimulus_path both come from the spreadsheet used for the trials loop.

	A	B	C	D	E	F	G
1	block_number	condition	onset_latency	stimulus_type	stimulus_path	blocks.thisRepN	blocks.thisTrialN
2	1	short	1.5	green	green_square.png	0	0
3	1	short	1.5	green	green_square.png	0	0
4	1	short	1.5	green	green_square.png	0	0
5	1	short	1.5	red	red_square.png	0	0
6	1	short	1.5	red	red_square.png	0	0
7	1	short	1.5	green	green_square.png	0	0
8	1	short	1.5	red	red_square.png	0	0
9	1	short	1.5	green	green_square.png	0	0
10	1	short	1.5	red	red_square.png	0	0
11	1	short	1.5	red	red_square.png	0	0
12	1	short	1.5	green	green_square.png	0	0

Note: This is why it's important to adequately label the experimental conditions in the spreadsheets used to control loops. Even if not functionally necessary, this will make analysis much easier down the road.

In addition to the variable names passed through loops, the data file contains a large number of columns generated by PsychoPy. These too correspond with variable names + values for a given trial. For example, key_resp.keys and key_resp.rt, which store the registered key and RT for the key_resp component in each trial, are also variable names that can be accessed in other components using $key_resp.keys and $key_resp.rt.

O	P	Q	R	S	T	U	V	W
oss.stoppe	image.starte	image.stopp	key_resp.keys	key_resp.rt	key_resp.sta	key_resp.sto	participant	session
2.9065964	13.3733954	None	left	0.375610114	13.3733954	None		
4.7908105	15.2741767	None	left	0.342219881	15.2741767	None		
5.6577127	17.1411233	None	left	0.308948407	17.1411233	None		
3.4921398	18.9755055	None	right	0.296839863	18.9755055	None		
0.3088362	20.7923546	None	right	0.358842361	20.7923546	None		
2.1924563	22.6760023	None	left	0.320063835	22.6760023	None		
4.0434819	24.5269017	None	right	0.381548789	24.5269017	None		
5.9436926	26.4270902	None	left	0.281342086	26.4270902	None		
7.7440403	28.2273524	None	right	0.303573896	28.2273524	None		
9.5782975	30.0615992	None	right	0.31417084	30.0615992	None		
1.4118895	31.8786647	None	left	0.342117389	31.8786647	None		

*Important: For any response component, component.rt will correspond with the RT from the **onset of that component.** If stimulus onset ≠ the onset of the response component, or you're interested in measuring response latency from some other point, it is critical that you adjust your RTs accordingly.*

Part 5: Processing RT Data with Pandas for Python (Single-subject)

Description: We'll now use Python and Pandas to process the RT data for a single subject. This example will use the file *S1_example_experiment_2023-10-23_11h54.26.543.csv* from example_experiment/data, but feel free to use your own data collected from the example experiment.

Documentation for Pandas: https://pandas.pydata.org/docs/user_guide/index.html

Here are the steps we'll follow:

0. Import dependencies and configure Pandas options.
1. Import the .csv data file into a Pandas data frame.
2. Clean up the data frame.
3. Filter the data.
4. Compute mean RTs and stats.

Note: While not included here, in the demos we'll **print the data frame to the console** at every step to confirm that the script is functioning properly. This is highly recommended when scripting yourself.

0. Let's begin by **importing the modules** we'll use to process the data. This includes Pandas, which we'll import per convention as pd, as well as the *ttest_ind* function from scipy.stats, used for computing independent-samples t-tests. For more on this function, see: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html.

```
import pandas as pd
from scipy.stats import ttest_ind
```

Additionally, we'll configure Pandas so that it doesn't truncate data frames when printed to the console. This isn't strictly necessary, but it will make reading them easier.

```
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

1. Now we can **import the RT data into a Pandas data frame.** To do this, we'll first set a variable for the relative path to the data file and then read the data into a variable called *rawData*.

```
dataPath = './sample_data/S1_example_experiment_2023-10-23_11h54.26.543.csv'

rawData = pd.read_csv(dataPath)
```

Note that depending on the delimiter for your .csv file, you may need to specify the delimiter to Pandas using the delimiter parameter. For example, delimiter=';' tells Pandas to look for a semicolon delimiter.

2. Next, we'll **clean up the data frame**. Since PsychoPy provides a large number of columns we don't need for analysis, let's first select only the columns we're interested in. Here, this will be the participant, the condition, the response key, and the RT for each trial.

```
rawData = pd.DataFrame(rawData, columns = ['participant', 'condition',
'key_resp.keys', 'key_resp.rt'])
```

We can also rename the columns to more user-friendly labels.

```
newLabels = {
    'participant': 'subj',
    'condition': 'cond',
    'key_resp.keys': 'resp',
    'key_resp.rt': 'RT'
    }

rawData = rawData.rename(columns = newLabels)
```

Notice that here the *rename()* method doesn't mutate rawData, but instead returns a new data frame, so we need to assign this new data frame to a variable.

We will also remove the first and last row in the data frame, since those aren't trials.

```
rawData = rawData.loc[1:44, :]
```

Note that this is an especially easy case, since the only non-trial rows are the first and last. However, in some cases you may need to filter (using the techniques below) to remove non-trial rows.

At this point, the data frame looks like this:

	subj	cond	resp	RT
1	S1	high	space	0.394209
2	S1	high	space	0.386102
3	S1	catch	NaN	NaN
4	S1	high	space	0.350280
5	S1	low	space	0.416333
6	S1	low	space	0.384099
7	S1	low	space	0.368574
8	S1	high	space	0.297284
9	S1	low	space	0.430557
10	S1	high	space	0.309573
11	S1	high	space	0.344251
12	S1	low	space	0.394874
13	S1	catch	NaN	NaN

3. It's now time to **filter the data**. Recall that when analyzing RTs, we need to remove any trials where the participant did not respond correctly. For this simple reaction design, filtering is particularly straightforward, since the correct response is always just *respond*. This means we only need to remove (i) catch trials and (ii) trials where the participant didn't respond at all.

   We'll start by defining condition variables to use in filtering.

```
notCatch = rawData.cond != 'catch'
response = rawData.resp == 'space'
```

   After which we can define a new data frame, *filtData*, which only contains the data we want for analysis.

```
filtData = rawData[notCatch & response]
```

4. With the data filtered, we can now **compute mean RTs and statistics** between conditions. To do this, we'll first define condition variables for both the high opacity and low opacity conditions.

```
condHigh = filtData.cond == 'high'
condLow = filtData.cond == 'low'
```

   This allows us to define arrays (called 'series' in Pandas) of the RTs for both conditions.

```
highRTs = filtData[condHigh].RT
lowRTs = filtData[condLow].RT
allRTs = filtData.RT
```

   We can then compute the means for each array using the *mean()* method. Let's also print these values out to the console.

```
print('mean RT (high opacity): ', highRTs.mean())
print('mean RT (low opacity): ', lowRTs.mean())
print('mean RT (both conditions): ', allRTs.mean())
```

Finally, we can use the *ttest_ind()* function to compute an independent-samples t-test between the high and low opacity conditions. Notice that here we'll also set equal_var to false to perform a Welch's t-test, which doesn't assume equal variance.

```
statsResults = ttest_ind(highRTs, lowRTs, equal_var= False)
print(statsResults)
```

Part 6: Processing RT Data with Pandas for Python (Group-level)

Description: In this part, we'll build out the single-subject pipeline to process and analyze data at the group level. Here are the steps we'll follow:

0. Import dependencies and configure Pandas options.
1. Specify the data files to import.
2. Create data frames to store group data.
3. Loop over every subject file, importing, cleaning up, and filtering (as in part 5).
4. Compute group mean RTs and stats.

0. As with the single-subject pipeline, we'll start by **importing modules and configuring pandas** to prevent console truncation. The only addition here is the *listdir()* function from *os*. Otherwise, everything should look familiar from part 5.

```
import pandas as pd
from scipy.stats import ttest_ind
from os import listdir

pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

1. Now we need a **list of the files to import.** To get this, we'll first define a path to the directory containing the data files, using *listdir()* to generate a list of all the files in that path.

```
dataPath = './sample_data/'
fileList = listdir(dataPath)
```

At this point, *fileList* will include all the files in sample_data, but we only want the .csv files. We can make sure to include only .csv files with a list comprehension.

```
dataFiles = [file for file in fileList if file.split('.')[-1] == 'csv']
```

2. Next, rather than importing a .csv file directly into an object for group-level data, we're going to **create empty data frames** to contain group level data. (In the next step, we'll then fill these data frames one subject at a time.)

   In this example, we're going to create two group-level dataframes. The first, *groupDataMeans*, will contain entries for **mean RTs** for each condition/subject. This will be used for performing t-tests between conditions, as well as creating bar graphs and box plots in part 7.

   The second data frame, *groupDataAll*, will contain **every individual trial** for every subject, with error trials filtered out. This will be used for creating histograms and scatter plots in part 7, and could also be used for linear mixed models in the optional part 9.

```
groupDataMeans = pd.DataFrame(columns = ['subj', 'cond', 'RT'])
groupDataAll = pd.DataFrame(columns= ['subj', 'cond', 'resp', 'RT'])
```

3. Now that we have a list of data files (§1) and empty group-level data frames (§2), we can **loop over each data file** to fill in those data frames. As there's a lot going on in this loop, we'll first look at it in full before walking through piece-by-piece.

```
rowIndex = 0

for file in dataFiles:
    ##3A. read data into pandas df
    rawData = pd.read_csv(dataPath + file)

    ##3B. clean up the dataframe

    #select only the columns we need
    rawData = pd.DataFrame(rawData, columns = ['participant', 'condition',
                  'key_resp.keys', 'key_resp.rt'])

    #rename those columns to more user-friendly labels
    newLabels = {
        'participant': 'subj',
        'condition': 'cond',
        'key_resp.keys': 'resp',
        'key_resp.rt': 'RT'
        }

    rawData = rawData.rename(columns = newLabels)

    #variable for the name of this subject (will use later)
    thisSubject = rawData.loc[1, 'subj']

    #remove first and last row (since those are not trials)
    rawData = rawData.loc[1:44, :]
```

```
#quality control: verify that we have the expected number of trials
expectedTrialCount = 44
trialCount = len(rawData)

if trialCount != expectedTrialCount:
    print('unexpected number of trials for subject', thisSubject)

##3C. filter the data

#condition variable for not a catch trial
notCatch = rawData.cond != 'catch'

#condition variable for response registered
response = rawData.resp == 'space'

#remove catch trials
filtData = rawData[notCatch & response]

##3D. add data from every trial to groupDataAll
groupDataAll = pd.concat([groupDataAll, filtData])

##3E. calculate mean RTs

#condition variable for each experimental condition
condHigh = filtData.cond == 'high'
condLow = filtData.cond == 'low'

#mean RT for each condition
highMeanRT = filtData[condHigh].RT.mean()
lowMeanRT = filtData[condLow].RT.mean()

##3F. add to groupData df: [subject, condition, mean RT]

#high condition
groupDataMeans.loc[rowIndex] = [thisSubject, 'high', highMeanRT]
rowIndex += 1

#low condition
groupDataMeans.loc[rowIndex] = [thisSubject, 'low', lowMeanRT]
rowIndex += 1
```

Let's talk about what this loop does. You may have noticed that many of the steps are familiar from part 5. This is because the loop performs much of the processing we've already discussed for single-subject data, except now it repeats that processing for multiple subjects.

Everything involved in reading in and cleaning up the *rawData* data frame proceeds much as it did in part 5, with the one addition being that we verify that the imported data has the expected number of trials (§3B).

```
    expectedTrialCount = 44
    trialCount = len(rawData)

    if trialCount != expectedTrialCount:
        print('unexpected number of trials for subject', thisSubject)
```

Since we may be importing a large number of data files, this is a quick way to catch if something has gone wrong.

Next, we filter the data as in the single-subject pipeline (§3C). After that, we can add the full data for this subject to the *groupDataAll* data frame, which we use to store every (correct) trial for every subject. We'll do this using the *pd.concat()* function.

```
    groupDataAll = pd.concat([groupDataAll, filtData])
```

The next step is to compute the mean RTs, again just like the single-subject example (§3E).

This only leaves adding these mean RTs to the *groupDataMeans* dataframe, which stores the mean RTs for every subject for each condition. Since we only have two conditions to add, we'll just do it manually, row-by-row. This is where the *rowIndex* variable, which we introduced before the start of the loop, comes in. We'll use this variable to keep track of the next row number for the *groupDataMeans* data frame, adding one to *rowIndex* every time we add a new row to *groupDataMeans*. That way, the next row in *groupDataMeans* can always be assigned using a statement like *groupDataMeans.loc[rowIndex] = . . .* We'll do this for both the high and low opacity conditions.

```
    #high condition
    groupDataMeans.loc[rowIndex] = [thisSubject, 'high', highMeanRT]
    rowIndex += 1

    #low condition
    groupDataMeans.loc[rowIndex] = [thisSubject, 'low', lowMeanRT]
    rowIndex += 1
```

4. Finally, now that we have a data frame containing the mean RTs for each condition for every subject, we can **compute stats.** Much like the single-subject stats, we can first define conditional variables for each experimental condition.

```
condHighGroup = groupDataMeans.cond == 'high'
condLowGroup = groupDataMeans.cond == 'low'
```

And then use these condition variables to get the series of mean RTs for each condition (as well as both conditions).

```
highRTsGroup = groupDataMeans[condHighGroup].RT
lowRTsGroup = groupDataMeans[condLowGroup].RT
allRTsGroup = groupDataMeans.RT
```

That leaves printing out the mean RTs to the console . . .

```
print('group mean RT (high opacity): ', highRTsGroup.mean())
print('group mean RT (low opacity): ', lowRTsGroup.mean())
print('group mean RT (both conditions): ', allRTsGroup.mean())
```

. . . and performing a t-test.

```
statsResults = ttest_ind(highRTsGroup, lowRTsGroup, equal_var= False)
print(statsResults)
```

Part 7: Visualizing RT Data with Matplotlib

Description: Now that we've processed and analyzed the data, it's time to visualize it. Here we'll use the matplotlib library for Python to create simple histograms, scatterplots, bar graphs, and box plots, both for single-subject and group-level data. The documentation for matplotlib can be found here: https://matplotlib.org/stable/index.html.

0. Let's begin by **importing the required modules.** Here we'll use matplotlib.pyplot, imported as plt per convention, as well as the sem function from scipy.stats, which we'll use to compute the standard error of the mean for adding error bars to our bar graphs.

   For now, we'll continue to work from the single-subject script developed in part 5, adding new code to the end of what we've already written. Later on in this section, we'll visualize group-level data, extending the part 6 script.

   In total, we need to import the following:

```
import pandas as pd
import matplotlib.pyplot as plt

from scipy.stats import ttest_ind, sem
```

1. Let's first visualize the distribution of RTs using a **histogram.** In its simplest form, we could simply run . . .

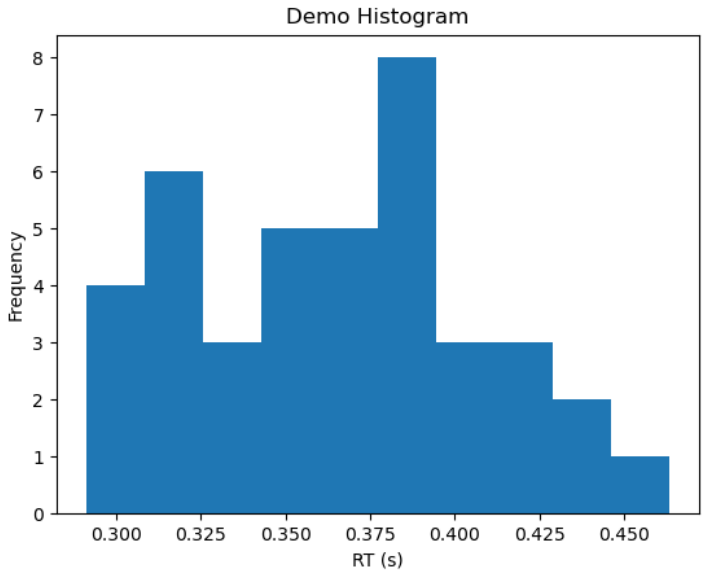
```
plt.hist(allRTs)

plt.show()
```

   . . . using the *plt.hist()* to draw a histogram and *plt.show()* to show it to the screen. However, a slightly nicer plot might use the *bins* parameter to adjust the number of bins in the histogram, *plt.xlabel()* and *plt.ylabel()* to add labels to the x and y axis, and *plt.title()* to add a title to the plot.

```
plt.hist(allRTs, bins = 10)
plt.ylabel("Frequency")
plt.xlabel("RT (s)")
plt.title("Demo Histogram")
plt.show()
```

Which generates a simple histogram.



While fairly low-resolution at only 44 trials, the skewed distribution characteristic of RT data is still clearly visible, rising quickly with a longer tail.

2. Next, let's make a **scatter plot.** We'll use the *plt.scatter()* function, which takes a list/array/series of x coordinates as its first parameter and list/array/series of y coordinates as its second parameter. The simplest version of a scatter plot would look like this . . .

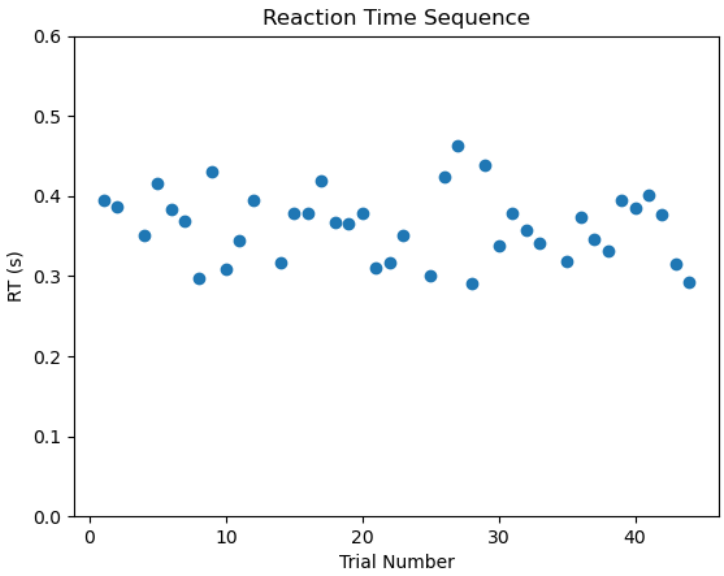
```
plt.scatter(allRTs.index, allRTs)

plt.show()
```

. . . using *allRTs.index* (trial number) as our x values and RTs for the y values. However, we can again make the plot a bit nicer by adding axis labels and a title. We'll also use *plt.ylim()* to set the range displayed on the y axis, so the data is a bit more contextualized.

```
plt.scatter(allRTs.index, allRTs)
plt.ylim(bottom=0, top = .6)
plt.ylabel("RT (s)")
plt.xlabel("Trial Number")
plt.title("Reaction Time Sequence")
plt.show()
```

This gives us a simple scatter plot.



Perhaps a bit unexpectedly, RTs are quite flat as the experiment progresses.

3. Time for a **bar graph.** We're going to start by using the *plt.subplots()* function, which returns objects for both the figure and axes. While our plot will only be made up of a single subplot, passing arguments to *plt.subplot()* can create plots with multiple subplots. For more, see: https://matplotlib.org/stable/users/explain/axes/axes_intro.html.

   We'll pass three parameters to the *ax.bar()* function. First is a list of the x coordinates for the center of our bars. This is largely arbitrary and will only affect how the plot looks. The second parameter is a list of the y coordinates (i.e., height) of the bars. In our case, these are the mean RTs for each condition. We'll also tell it how wide we want the bars using the *width* parameter. Again, this is largely arbitrary and merely cosmetic.

```
fig, ax = plt.subplots()

bars = ax.bar([.5,1.5], [highRTs.mean(), lowRTs.mean()], width=.4)

plt.show()
```

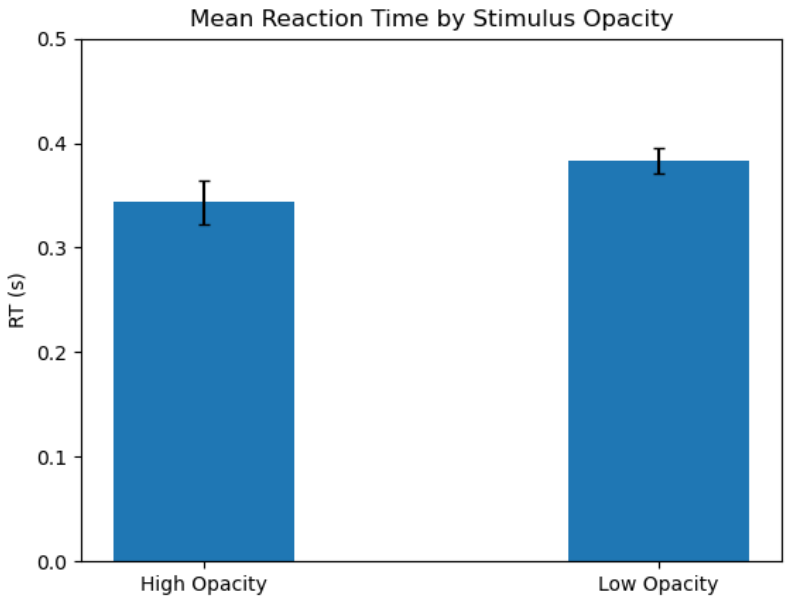
   As with the previous plots, we'll add some labels and a title. Notice that here we're using different functions than above, which now use the *ax* object. Notice too that we label each bar by first setting x ticks to only the center of each bar, and then labeling each of those ticks.

```
ax.set_ylabel('RT (s)')
ax.set_ylim(top=.5)
ax.set_title('Mean Reaction Time by Stimulus Opacity')
ax.set_xticks([.5,1.5])
ax.set_xticklabels(["High Opacity", "Low Opacity"])
```

Finally, let's add error bars to the graph, using the conventional +/- 2 SEM. We can calculate the SEM using the *sem()* function imported above, and then add error bars with *ax.errorbar()*. The first two parameters the function takes, x and y, are just the x and y coordinates of the center of the error bar. In this case, those are the very same values we used to draw the main bars. Yerr takes a list of the y values of the error bars for each main bar, in this case 2*SEM. We'll also make a few cosmetic choices, setting the capsize, color, and preventing matplotlib from drawing a connection between them by setting *fmt = 'none'*.

```
highSEM = sem(highRTs)
lowSEM = sem(lowRTs)
ax.errorbar(x = [.5,1.5],
            y = [highRTs.mean(), lowRTs.mean()],
            yerr = [2*highSEM, 2*lowSEM],
            capsize = 3,
            ecolor = "black",
            fmt = "none")
```

The output bar chart:



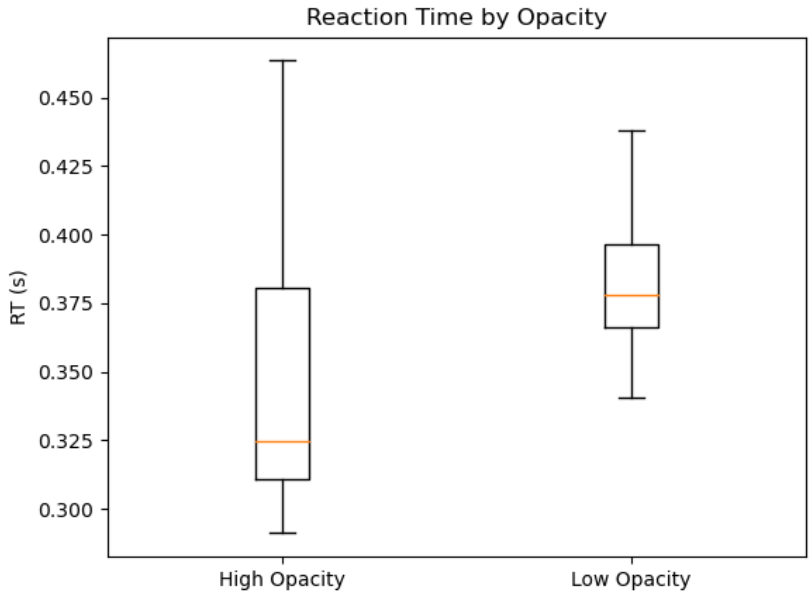
4. **Box plots.** Again we'll use *plt.subplots()* to return an axis object, which allows us to easily draw a box plot with *ax.boxplot()*. We'll give it a list of RTs for each condition, along with adding labels and a title.

```
fig, ax = plt.subplots()

box = ax.boxplot([highRTs, lowRTs])

ax.set_ylabel('RT (s)')
ax.set_title('Reaction Time by Opacity')
ax.set_xticklabels(['High Opacity', 'Low Opacity'])

plt.show()
```



Here we can see that, while the median (the colored horizontal line) is much lower for the high opacity condition the interquartile range (and +/- 1.5IQR) is noticeably wider.

5. Let's close out by discussing how to **extend the group-level script** (part 6) to also visualize that data. While this will of course depend on what you're interested in, here we'll use histograms and scatter plots to visualize raw data for individual trials, with bar and box plots used to visualize subject means.

   The code for the histogram and scatter plot is almost identical, with the key difference being that now we're using *groupDataAll* for the data.

```
#histogram
plt.hist(groupDataAll.RT, bins = 10)
plt.ylabel("Frequency")
plt.xlabel("RT (s)")
plt.title("Demo Histogram")
plt.show()

#scatter plot
plt.scatter(groupDataAll.index, groupDataAll.RT)
plt.ylim(bottom=0, top = .6)
plt.ylabel("RT (s)")
plt.xlabel("Trial Number")
plt.title("Reaction Time Sequence")
plt.show()
```

   The code for the bar graph and box plot will also look quite similar, with the major difference now that we're plotting *highRTsGroup* and *lowRTsGroup*.

```
#bar graph
fig, ax = plt.subplots()

bars = ax.bar([.5,1.5], [highRTsGroup.mean(), lowRTsGroup.mean()],
width=.4)

ax.set_ylabel('RT (s)')
ax.set_ylim(top=.5)
ax.set_title('Mean Reaction Time by Stimulus Opacity')
ax.set_xticks([.5,1.5])
ax.set_xticklabels(["High Opacity", "Low Opacity"])

#standard error of the mean for error bars
highSEM = sem(highRTsGroup)
lowSEM = sem(lowRTsGroup)
ax.errorbar(x = [.5,1.5],
            y = [highRTsGroup.mean(), lowRTsGroup.mean()],
            yerr = [2*highSEM, 2*lowSEM],
            capsize = 3,
            ecolor = "black",
            fmt = "none")

plt.show()
```

```
#box plot
fig, ax = plt.subplots()

box = ax.boxplot([highRTsGroup, lowRTsGroup])

ax.set_ylabel('RT (s)')
ax.set_title('Reaction Time by Opacity')
ax.set_xticklabels(['High Opacity', 'Low Opacity'])

plt.show()
```

And that's it!

As a final word: When analyzing and reporting group-level data (e.g., in a journal article, thesis, or project report), it's probably necessary only to visualize group-level data, not anything at the level of individual subjects. Here we've gone over both primarily for illustrative purposes.

Part 8: Bonus Content

Description: If you've moved faster than the pace of the class, or you're looking to learn more advanced techniques, why not try out the challenge tasks. These are listed in order of difficulty, and can all be done independently of one another.

Challenge Task 1: Violin plots

Use matplotlib to draw a violin plot for group-level data. Syntactically, it will be most similar to the box plot. To get started, see:

- https://matplotlib.org/stable/gallery/statistics/boxplot_vs_violin.html
- https://matplotlib.org/stable/gallery/statistics/violinplot.html
- https://matplotlib.org/stable/gallery/statistics/customized_violin.html

Challenge Task 2: Statsmodels

Use the statsmodels library for Python (https://www.statsmodels.org/stable/index.html) to compute more advanced statistical testing of your choosing (e.g., mixedlm for a mixed linear model). For those used to R-style syntax, statsmodels.formula might be a good place to start: https://www.statsmodels.org/stable/api.html#statsmodels-formula-api

Challenge Task 3: Real-time and cumulative feedback in PsychoPy

Using the code component, add the following features to *example_experiment.psyexp*:

1. Immediate feedback that displays RT (to the nearest ms) if a participant responds and positive/negative feedback if they don't respond. For example, it could say "good!" if they're correct to withhold response (i.e., in a catch trial) and "whoops!" if they're incorrect to withhold a response (i.e., they miss the target stimulus). Notice the first steps for real-time feedback are already included in *example_experiment.psyexp*.

2. Cumulative feedback after all trials are completed, which tells participants: (i) their mean RT, (ii) their lowest RT, and (iii) how many target stimuli they missed.

3. A final routine that participants can only reach if they miss sufficiently few stimuli. Otherwise, the trials repeat. (similar to a training portion of an actual experiment)

   Tip: to break out of a PsychoPy loop, try setting *loopName.finished = True*.

Remember that components compile in the order that they're listed within a routine, so the stimulus/response components that need some variable(s) must be positioned below code components that define/manipulate those variables.