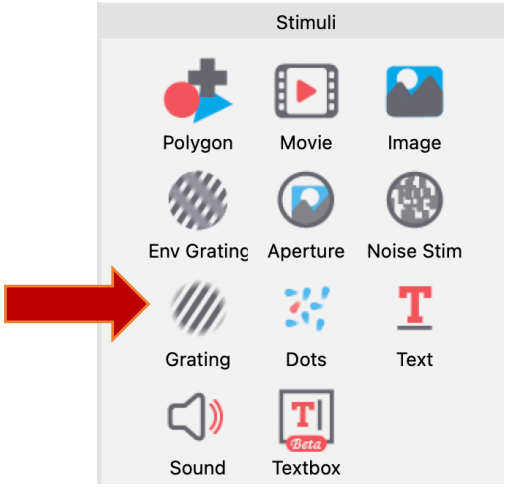
SDT Demos

Psychophysics: Theory and Application | 13-16 Nov., 2023

Contents

Part 1: PsychoPy Builder: Gratings and Staircase Loops

Description: We're going to begin by walking through two additional features of PsychoPy that are especially useful for visual psychophysics, the grating component and the "staircase" setting in loops. Together, these allow us to present neutral stimuli at the same thresholds for every participant.



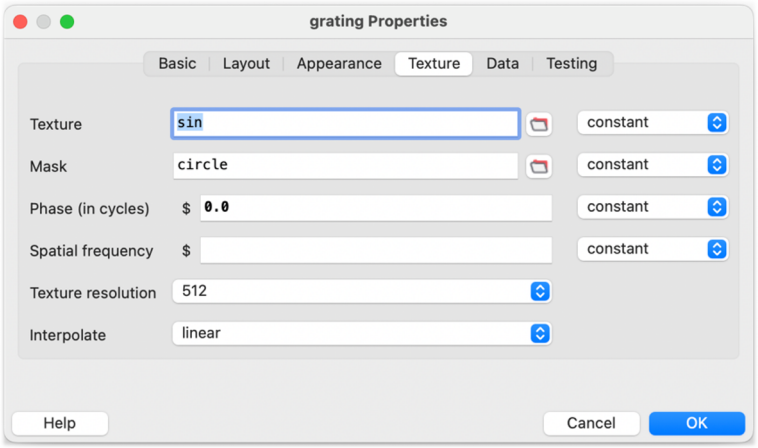
1. **The Grating Component:** This component allows you to present frequency gratings as visual stimuli, without having to generate any image files outside of PsychoPy. Using built-in gratings is a fast, easy way to generate stimuli, and PsychoPy provides a large number of options for changing their appearance. For a full list of features, see: https://www.psychopy.org/builder/components/grating.html

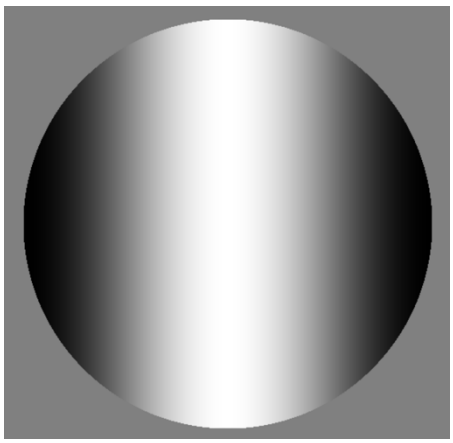
Here are some quick settings that will make the grating look better:

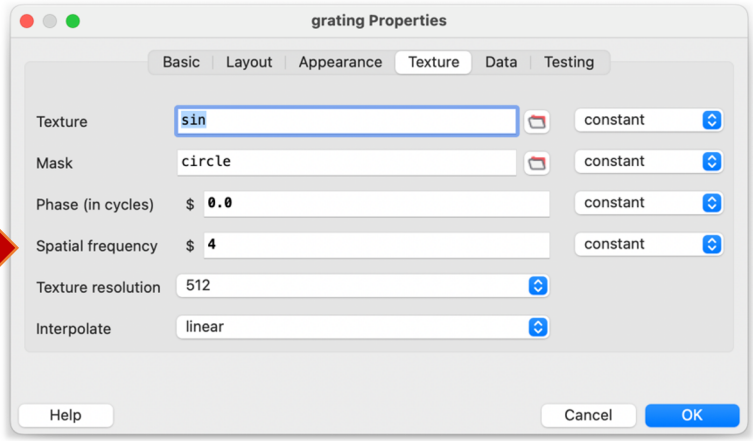
- Texture --> mask = circle (just type *circle*).
- Texture --> texture resolution = 512

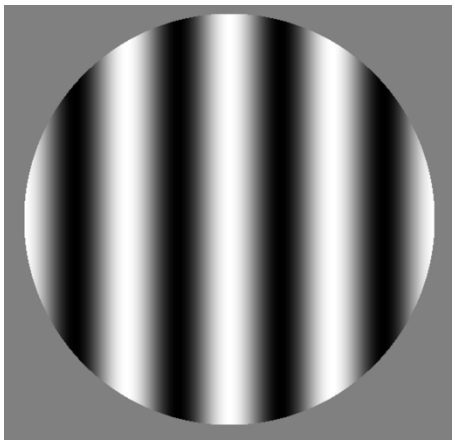
Other key features (examples below):

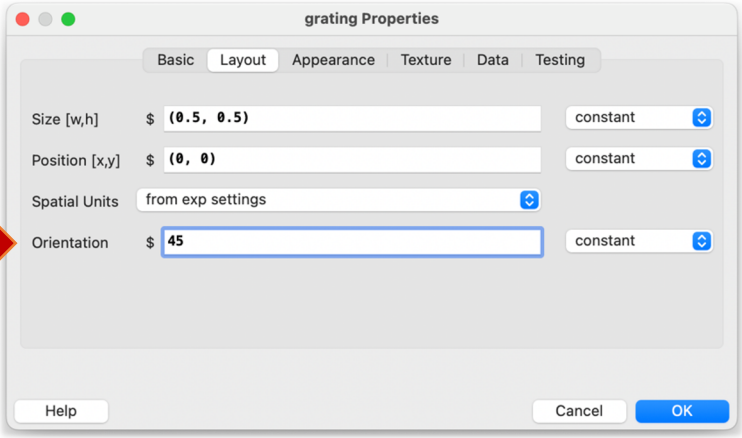
- Texture --> Spatial frequency: Number of cycles in the grating
- Layout --> Orientation: Rotation of grating (in degrees)
- Appearance --> Contrast: The contrast between the peak high and low values [0,1]

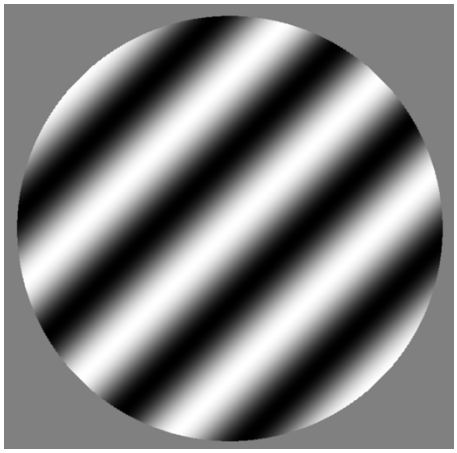

grating Properties
Basic Layout Appearance Texture Data Testing
Texture sin constant
Mask circle constant
Phase (in cycles) $ 0.0 constant
Spatial frequency $ constant
Texture resolution 512
Interpolate linear
Help Cancel OK

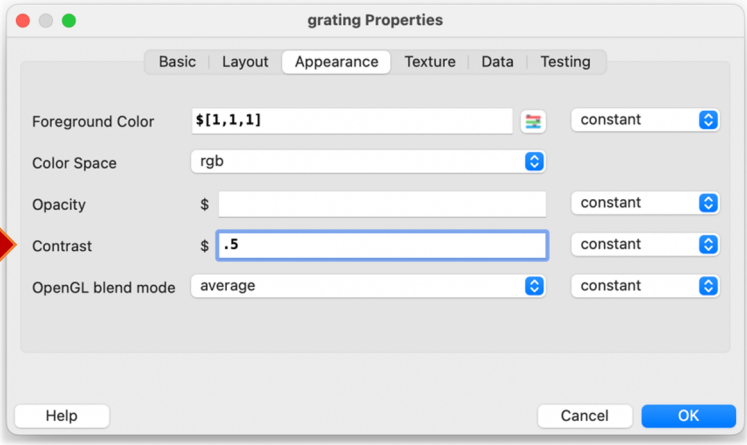


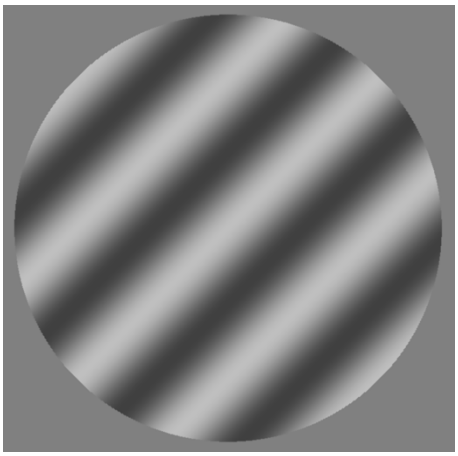

grating Properties
Basic Layout Appearance Texture Data Testing
Texture sin constant
Mask circle constant
Phase (in cycles) $ 0.0 constant
Spatial frequency $ 4 constant
Texture resolution 512
Interpolate linear
Help Cancel OK




grating Properties
Basic Layout Appearance Texture Data Testing
Size [w,h] $ (0.5, 0.5) constant
Position [x,y] $ (0, 0) constant
Spatial Units from exp settings
Orientation $ 45 constant
Help Cancel OK




grating Properties
Basic Layout Appearance Texture Data Testing
Foreground Color $[1,1,1] constant
Color Space rgb
Opacity $ constant
Contrast $ .5 constant
OpenGL blend mode average constant
Help Cancel OK

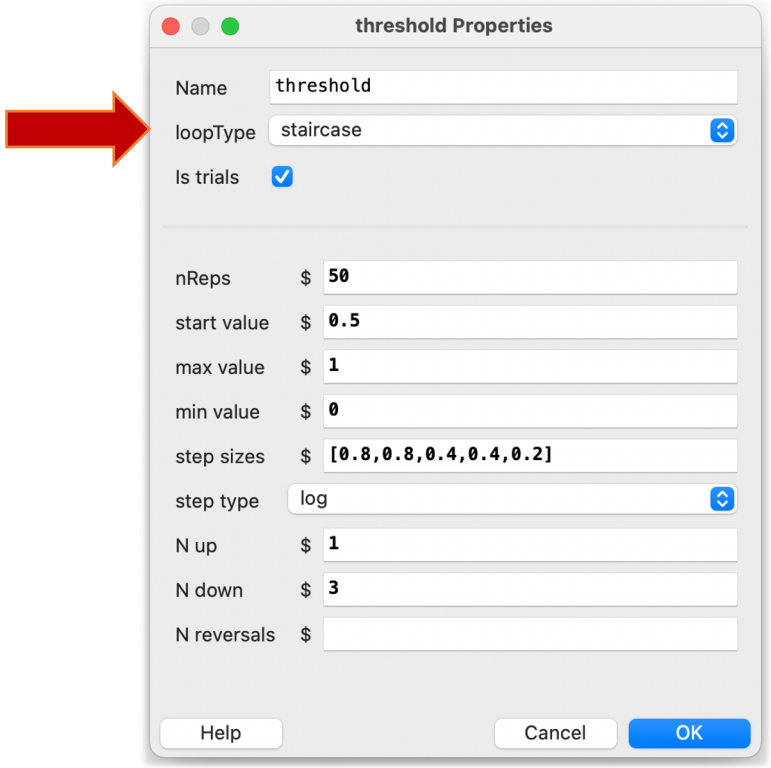


2. **Staircase Loops:** The "staircase" setting for loops lets you use an up-down adaptive staircase to approximate the visual detection thresholds of each participant.

   Important: The staircase loop will create an output variable called **level**, which can be used to adjust (e.g.) the contrast of a stimulus just by typing *level* in the contrast field.

   The parameters on staircase loops are themselves easy to set, and for now we'll stick with the default settings.

   Notice that this is set to **1-up/3-down**. It takes three correct responses in a row to move a step down the staircase, but only one incorrect answer to move a step up. This corresponds with around an 80% threshold.

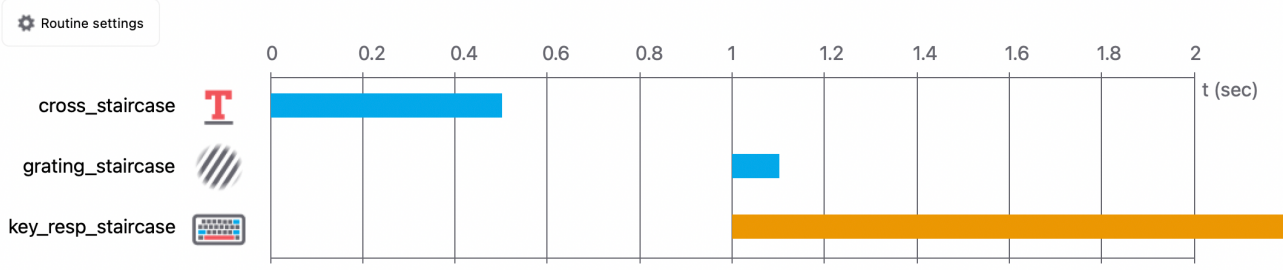


1-up/4-down --> 84.1%
1-up/3-down --> 79.4%
1-up/2-down --> 70.7%
1-up/1-down --> 50%

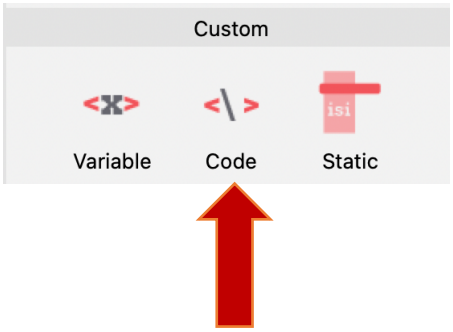
If N reversals is set, it will only conclude the loop after both that number of reversals AND the number of repetitions (nReps) are reached.

Note also that if multiple step sizes are given, the loop will only conclude after all those steps have been exhausted, which **requires a reversal for every change in step size.**

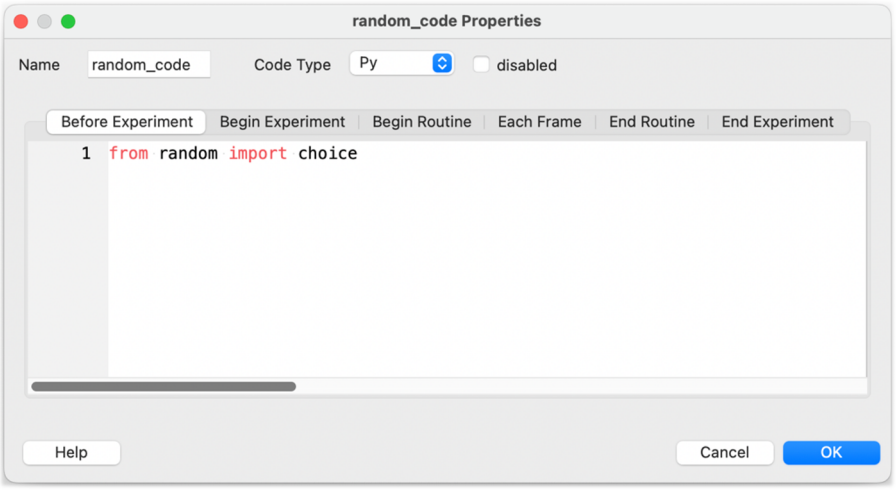
Most of the setup for a staircase loop in PsychoPy isn't actually done in the loop itself, but rather the other components you put around it. Here, we're going to create a routine that starts with a 500ms fixation cross, with stimulus onset 500ms after the end of the cross. The stimulus will be a grating with the following settings: duration = .1s; size = (0.2, 0.2) with spatial units in "height;" orientation = 45; contrast = *level* (the output of the staircase loop); opacity = .1; mask = *gauss*; spatial frequency = 8; texture resolution = 512. We'll also add a response component to this routine, which begins with stimulus onset and only ends when a response is registered. Together, the routine should look like this:



Now comes the tricky bit. We need to have the loop include some kind of response task and tell the response component which responses are correct and which aren't. In this case, the task will be to **determine whether the grating is presented in the right portion of the screen or the left portion of the screen.** Participants will respond with the right arrow for the right side, and left arrow for the left side. To do this, we'll need to make use of the *code* component, which allows us to insert both Python and JavaScript segments into the compiled code.



First, it's important to set the code type to "Py."

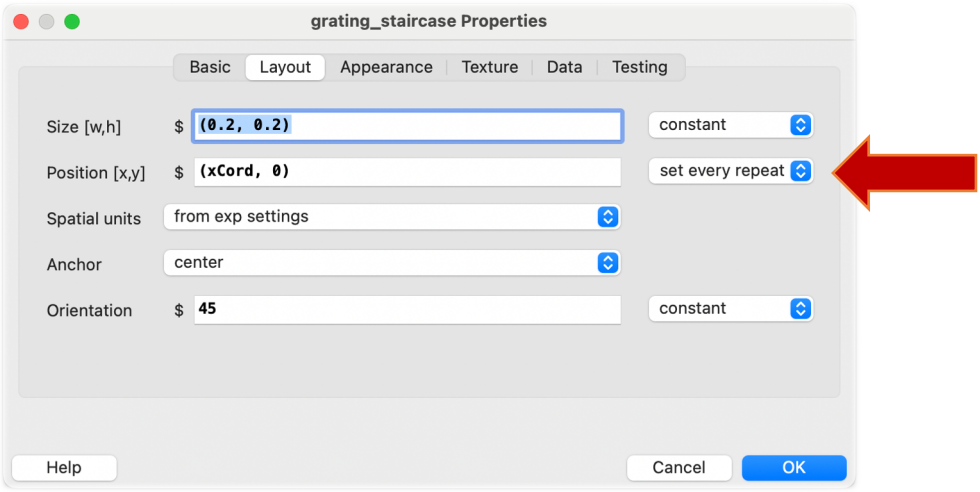


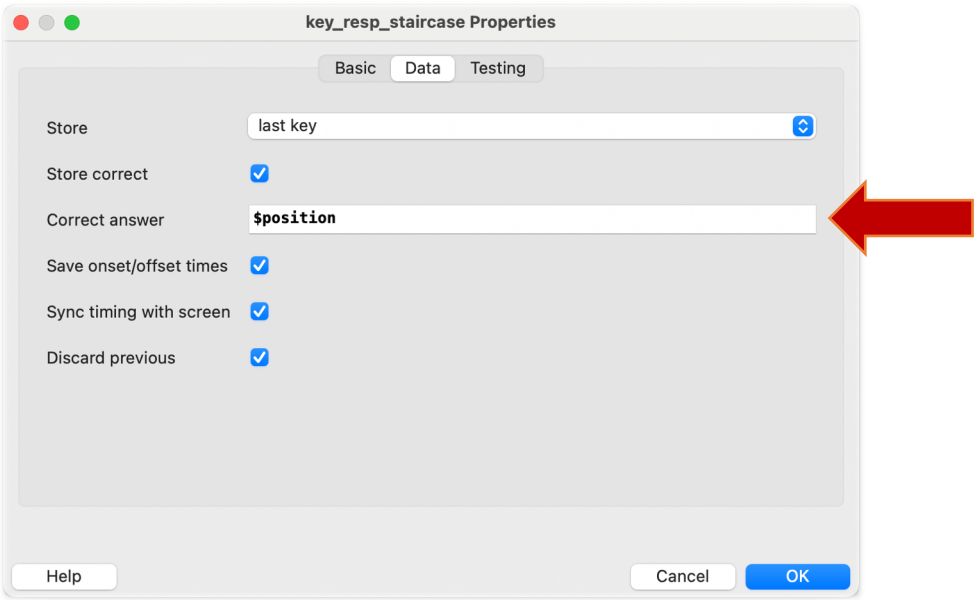
Note: You'll probably only ever need an experiment compiled into JavaScript if you're running it online. However, if you are, PsychoPy's automatic conversion from Python to JavaScript isn't great. It's usually a better idea to write the code segments in both languages yourself.

Now we're going to import the *choice* function from Python's random module. This will allow us to randomly choose the left or right side for each trial. Notice that, just like any other Python script, this goes before the experiment begins.

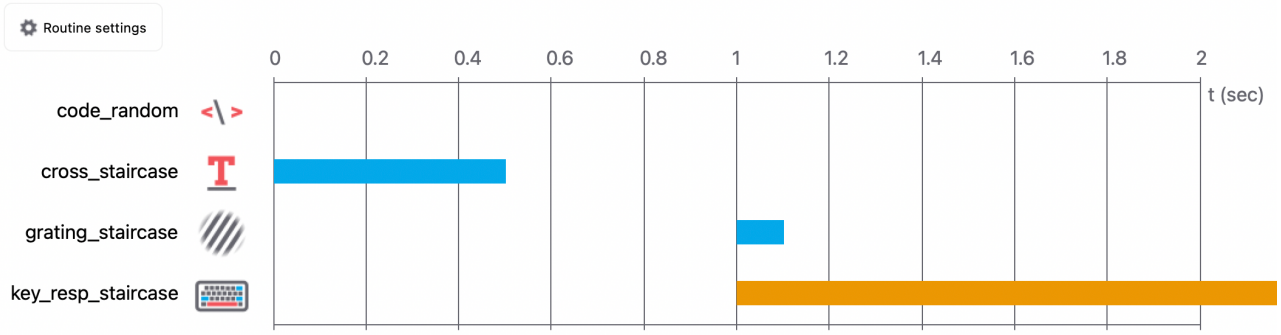
Next, at the beginning of the routine, we're going to create a variable called *position*, that will randomly be right or left for each trial. We'll then define a second variable, *xCord,* based on the position for that trial, which sets the x coordinate for the grating. An xCord of -.3 will shift the image to the left, and an xCord of .3 will shift the image to the right. Note also the default value 0, which isn't accessed in correct operation.

Finally, we can add these variables to control the grating and response components.





Finally, make sure that this code component is positioned before any components that need the variables that it computes. You can do this by right clicking the component and selecting “move to top.”



Now the staircase loop is ready to calculate the ≈80% threshold for each participant individually.

Part 2: PsychoPy Builder Practice

Description: Now we'll practice by building a simple experiment that utilizes an adaptive staircase.

This experiment should have the following properties:

1. The **task** should be the same as in part 1: Respond whether the stimulus grating was presented on the left or right portion of the screen. An adaptive staircase will first set the contrast of the grating for an approximately 70% threshold, after which the main experiment will use this fixed contrast value.

2. Both the staircase and main trials should use a **2-alternative forced-choice design**, where participants must respond left or right before the next trial can begin. Both the staircase and main trials should use a fixation cross of 500ms and then a fixed 500ms latency period before stimulus onset.

3. Constant values for the **grating** (both staircase and main trials):

   - Size: (0.2, 0.2)
   - Orientation: 45
   - Opacity: .1
   - Texture: sin
   - Mask: gauss
   - Spatial frequency: 8
   - Texture resolution: 512

4. Constant values for the **staircase loop**:

   - Repetitions: 50
   - Start value: .5
   - Max value: 1
   - Min value: 0
   - Step sizes: [.8,.4,.2,.1]
   - Step type: log
   - 1-up/2-down to approximate a 70% threshold

   Additionally, the duration of the grating stimulus should be fixed to .1 for the staircase.

5. The main trials should have **two conditions**, a short duration condition, in which the stimulus is presented for 100ms, and a long duration condition in which the stimulus is presented for 200ms. There should be 40 trials for each condition (20 left + 20 right), with a random duration and position for every trial.

6. Participants should receive a message that immediately precedes the main trials, informing them that these trials are about to begin.

Once the experiment is built, you can try it yourself.

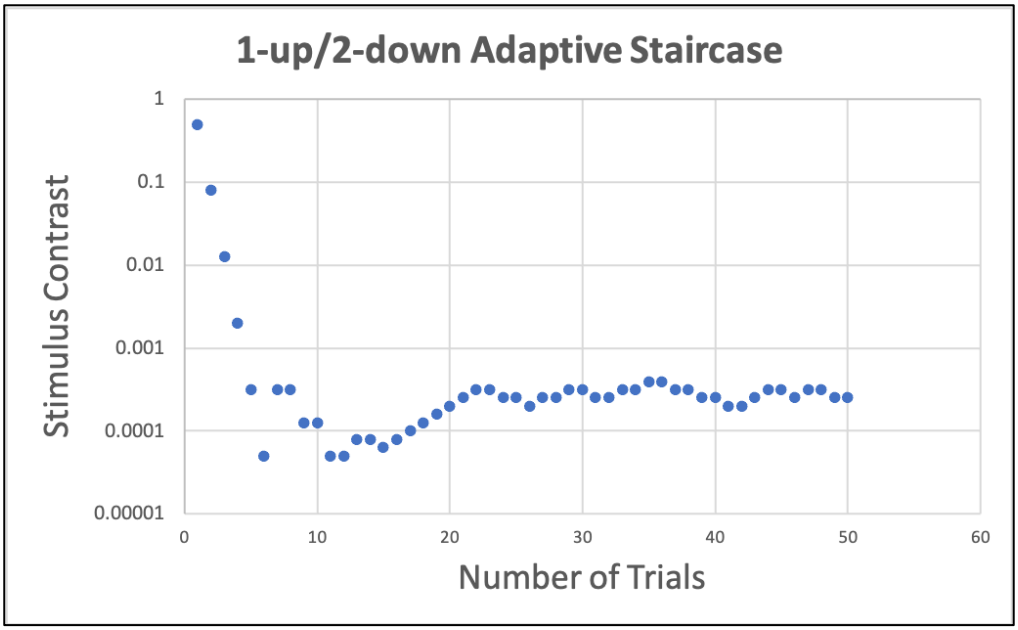
Part 3: Visualizing the Staircase

Description: Let's now take a quick look at what staircase loops actually do. Here we'll use the *demo_data_SDT.csv* file, but you can also use the data you collected in part 2.

The *threshold.stepSize* column records the pre-set step size for each trial, and the *threshold.intensity* column records the actual value passed by the staircase, which corresponds with the *level* variable. (Note that "threshold" in this case is just the chosen name for the staircase loop in the demo example.)

To visualize what the staircase did, we can quickly graph the *threshold.intensity* column as a scatter plot in Excel. Since the intensity decreased exponentially, it's easier to see its effects using a log scale on the y-axis.

C	D	E
threshold.thisTrialN	threshold.stepSize	threshold.intensity
0	0.8	0.5
1	0.8	0.07924466
2	0.8	0.012559432
3	0.8	0.001990536
4	0.8	0.000315479
5	0.8	5.00E-05
6	0.8	0.000315479
7	0.4	0.000315479
8	0.4	0.000125594
9	0.4	0.000125594
10	0.4	5.00E-05
11	0.2	5.00E-05
12	0.2	7.92E-05
13	0.1	7.92E-05
14	0.1	6.29E-05
15	0.1	7.92E-05
16	0.1	9.98E-05
17	0.1	0.000125594
18	0.1	0.000158114
19	0.1	0.000199054
20	0.1	0.000250594
21	0.1	0.000315479
22	0.1	0.000315479
23	0.1	0.000250594
24	0.1	0.000250594



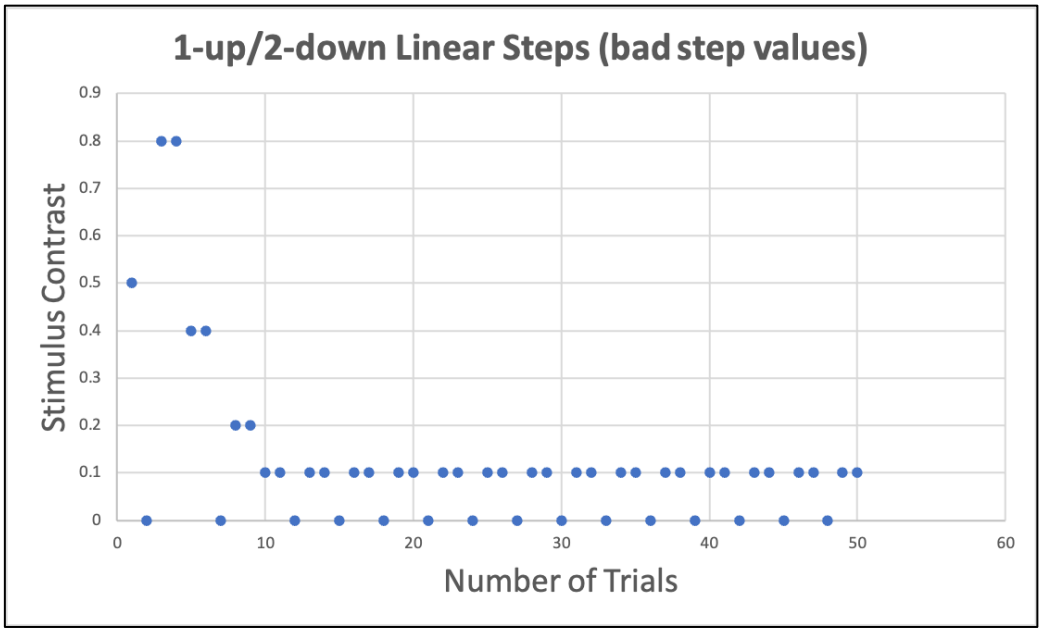
The value for each trial is computed recursively from the value for the previous trial. With N = trial number and s = step size, for steps down:

$$contrast_N = \frac{contrast_{N-1}}{10^s}$$

and for steps up:

$$contrast_N = (contrast_{N-1}) * 10^s$$

If the step type was set to "linear," the staircase would simply step up and down by +/- the current step size. Notice that this is clearly far too coarse-grained given the current step values, which would need to be substantially revised to work as intended:



Part 4: Processing SDT data using Pandas for Python

Description: We'll now walk through the steps required to calculate d' and criterion for a single subject. The examples will use the data file for S1 in the sample_data folder, but feel free to use your own data collected in part 2.

As a preliminary, we can import the modules we need. This time, it's just pandas and the norm object from scipy.stats, which we'll use in the formulas for d' and criterion.

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

1. Just like with RT data, we'll start by **opening the data file**.

```
#path to file folder + file name
dataPath = './sample_data/'
fileName = 'S1_example_staircase_2023-11-06_14h47.09.923.csv'

#read file into pandas
rawData = pd.read_csv(dataPath + fileName)
```

2. Next, we'll **clean up the rawData data frame.** Let's select only the columns we want and re-name the columns to make them more manageable.

```
#select only the columns we need
rawData = pd.DataFrame(rawData, columns = ['participant',
                                           'durationCondition',
                                           'position',
                                           'key_resp_main.keys'])

#rename those columns to more user-friendly labels
newLabels = {
    'participant': 'subj',
    'durationCondition': 'cond',
    'position': 'side',
    'key_resp_main.keys': 'resp'
    }

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

This time we've selected columns for the subject ('subj'), condition ('cond'), position of the stimulus ('side), and response ('resp'). If we were also analyzing RT data at the same time, we would want to include those columns at this stage too. But for this example, we'll only select the data for calculating d' and criterion.

Now we'll remove rows that don't correspond with trials. As we saw in the RT demos, how you do this depends largely on the structure of your data. In this case, a quick inspection reveals that for all non-trial rows, the cond column is given as NaN (not a number). This means we can remove all these using the notnull() method. We'll subsequently assign the new data frame to the variable trialData.

```
isTrials = rawData.cond.notnull()
trialData = rawData[isTrials]
```

We will also reset the index for the trialData (using drop=True to drop the old indices from the data frame entirely) and add one to every index so that they start from 1, not 0.

```
trialData = trialData.reset_index(drop=True)
trialData.index += 1
```

Notice that now, unlike when processing RT data, we **do not remove error trials.** This is because d' and criterion are calculated using hit rates and false alarm rates. Removing error trials would entirely defeat the purpose. This means that, if you were processing RT and SDT data with the same script, here you would have to make a second copy of trialData using the .copy() method to then use in the RT analysis.

3. At this point we can **calculate the SDT metrics** for the choice task. For this example, we'll define the left position as the "target" stimuli: responding "left" for a left = a hit; responding "right" for a right = a correct rejection; responding "left" for a right = a false alarm; and responding "right" for a left = a miss. In this case, the decision is arbitrary. There is no theory-motivated reason to select left over right for our task. (However, for other tasks, there might be.)

   To calculate d' and criterion, we first need the hit rate (HR) and false alarm rate (FAR) for each condition. To compute these, we'll start by defining variables for filtering data by (i) whether a target was presented, (ii) whether the participant responded that a target was presented, and (iii) the experimental conditions.

```
isTarget = trialData.side == 'left'
notTarget = trialData.side != 'left'

respTarget = trialData.resp == 'left'

condLong = trialData.cond == 'long'
condShort = trialData.cond == 'short'
```

   With these, we can then calculate the number of target and non-target trials for each condition (plus both conditions). While we already know how many *should* be there (20, 20, and 40 for both target and non-target), simply using the expected number without checking how many are actually there will lead to difficult-to-spot errors in the case that the number of trials is different than expected.

   Our strategy here will be to fill empty dictionaries with the trial count for both target and non-target trials, which will end up looking something like {'all' : 40, 'long' : 20, 'short' : 20}. The motivation behind this approach will become clear shortly.

   We'll use len() to quickly compute the number of trials for different conditions and stimuli.

```
targetTrials = {}
notTargetTrials = {}

targetTrials['all'] = len(trialData[isTarget])
notTargetTrials['all'] = len(trialData[notTarget])

targetTrials['long'] = len(trialData[isTarget & condLong])
notTargetTrials['long'] = len(trialData[notTarget & condLong])

targetTrials['short'] = len(trialData[isTarget & condShort])
notTargetTrials['short'] = len(trialData[notTarget & condShort])
```

Now we can check that the number of found trials matches the expected number. For a single subject, perhaps we could just print the found number to the console and verify manually. However, to make things easier for building out the script to handle multiple subjects, we'll instead perform the check in the code.

First, we'll hard-code in the number of expected target and non-target trials . . .

```
expectedTargets = {'all': 40, 'long': 20, 'short': 20}
expectedNotTargets = {'all': 40, 'long': 20, 'short': 20}
```

. . . and then loop over every condition to confirm that everything is as expected, printing a message if something isn't right.

```
conditions = ['all', 'long', 'short']
for cond in conditions:
    if targetTrials[cond] != expectedTargets[cond] or notTargetTrials[cond] != expectedNotTargets[cond]:
        print('Possible error with trial counts')
        print('Expected targets:', expectedTargets, 'Found targets:', targetTrials)
        print('Expected non-targets:', expectedNotTargets, 'Found non-targets:', notTargetTrials)
```

Note that we can easily test this check by changing a value in one of the expected dictionaries to something unexpected.

```
expectedTargets = {'all': 41, 'long': 20, 'short': 20}
expectedNotTargets = {'all': 40, 'long': 20, 'short': 20}
```

```
Possible error with trial counts
Expected targets: {'all': 41, 'long': 20, 'short': 20}
Found targets: {'all': 40, 'long': 20, 'short': 20}
Expected non-targets: {'all': 40, 'long': 20, 'short':
20} Found non-targets: {'all': 40, 'long': 20, 'short':
 20}
```

Now that we've confirmed that the number of found trials is as expected, we can calculate the hit and FA counts for each condition. We'll start by creating two new data frames, one that only contains hit trials and the other that only contains false alarm trials.

```
hitTrials = trialData[isTarget & respTarget]
faTrials = trialData[notTarget & respTarget]
```

With these, we can then fill up dictionaries for hit count and hit rate for each condition, just like with trial counts.

```
hitCount = {}
faCount = {}

hitCount['all'] = len(hitTrials)
faCount['all'] = len(faTrials)

hitCount['long'] = len(hitTrials.loc[condLong])
faCount['long']= len(faTrials.loc[condLong])

hitCount['short'] = len(hitTrials.loc[condShort])
faCount['short'] = len(faTrials.loc[condShort])
```

Note that we're using .loc[] here simply to suppress a re-indexing warning.

Now that we have the hit and FA counts for each condition, we're going to adjust any instance where the hit count = the number of target trials (HR = 1) or FA count = 0 (FAR = 0). In such cases, d' and criterion would be calculated to be infinite. The adjustment we'll use subtracts 1 from the hit count and adds 1 to the FA count. We'll also print a message to the console to inform us when an adjustment has been made.

Since our hit and FA counts are stored in dictionaries, it's easy to loop over conditions to quickly access all of them.

```
for cond in conditions:
    #adjust hit count
    if hitCount[cond] == targetTrials[cond]:
        hitCount[cond] -= 1
        print('Hit count for condition =', cond, 'adjusted to', hitCount[cond])
    #adjust fa count
    if faCount[cond] == 0:
        faCount[cond] = 1
        print('FA count for condition =', cond, 'adjusted to 1')
```

Now, finally, we can compute the hit and FA **rates**. Once again, we'll do this by filling dictionaries. Since our hit counts and target counts are already in similarly structured dictionaries, we can fill these dictionaries by looping over the conditions.

```
hitRate = {}
faRate = {}

for cond in conditions:
    hitRate[cond] = hitCount[cond]/targetTrials[cond]
    faRate[cond] = faCount[cond]/notTargetTrials[cond]
```

With HR and FAR in hand, the last step is to compute d' and criterion. To do this, we'll define functions to calculate each. Per convention, we'll place these functions at the top of the script.

```
#d' function
def dPrime(HR, FAR):
    result = norm.ppf(HR) - norm.ppf(FAR)
    return result

#criterion function
def criterion(HR, FAR):
    result = -.5*(norm.ppf(HR) + norm.ppf(FAR))
    return result
```

Notice that here norm.ppf(), denoted as *z* in the lectures, is the inverse of the cumulative distribution function of the standard normal distribution (a.k.a. the "percent point function" or "quantile function" for the normal distribution, or just "probit").

Now, finally, we can print out our findings to the console by looping over conditions.

```
#print HRs and FARs to console
for cond in conditions:
    print('HR', cond, '=', hitRate[cond])
    print('FAR', cond, '=', faRate[cond])
    print('\n')

#d' and criterion
for cond in conditions:
    print("d'", cond, '=', dPrime(hitRate[cond], faRate[cond]))
    print('criterion', cond, '=', criterion(hitRate[cond], faRate[cond]))
    print('\n')
```

And that's it!

4. As a bonus step, let's **format trialData so that we can use it with the SDT Kamu app** (see part 7). This means encoding target stimuli as 1 and non-target stimuli as 0.

   To start, we'll make a copy of trialData so that our changes don't affect it. Note that because all we need here is trialData, this step can be done *before* any of step 3.

   ```
   appData = trialData.copy()
   ```

   We'll then use the .replace() method to encode left (our target) as 1 and right (our non-target) as 0.

   ```
   appData = appData.replace({'left': 1, 'right': 0})
   ```

Then all we have to do is save this data frame to a new .csv file (notice that the app takes semi-colon delimiters).

```
pathOut = './exported_data/'
fileOut = 'S1.csv'

appData.to_csv(pathOut + fileOut, sep=';')
```

Part 5: Group-level analysis using Pandas for Python

Description: We're now going to expand the single-subject script to handle data from multiple subjects. Much like the group-level RT script discussed in the RT demo, our strategy will be to loop over each subject, computing d' and criterion subject-by-subject and adding them to group-level data frames.

0. Let's **set up the script** by importing necessary modules, configuring pandas to prevent truncating in the console, and defining d' and criterion functions. All these should be familiar from part 4 and/or the RT demos.

```
import pandas as pd
import matplotlib.pyplot as plt
from os import listdir
from scipy.stats import norm, ttest_ind

#prevent from truncating dataframes when printing to console
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)

#d' function
def dPrime(HR: float, FAR: float) -> float:
    result = norm.ppf(HR) - norm.ppf(FAR)
    return result

#criterion function
def criterion(HR: float, FAR:float ) -> float:
    result = -.5*(norm.ppf(HR) + norm.ppf(FAR))
    return result
```

Notice that here the dPrime and criterion functions include typing, indicating that the HR and FAR parameters should be type float, with the function then returning type float. This typing isn't necessary (Python doesn't even enforce it), but it may help as a reminder of how the function works.

1. With the set-up out of the way, we can (i) **define a path to the data files** we want to process, (ii) **list the files** in those locations using listdir(), (iii) s**elect only .csv files,** and (iv) **sort those files** by name.

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

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

dataFiles.sort()
```

2. Now let's **define the empty data frames we'll fill with group data.** The first of these, *groupDataSDT,* will contain the HR, FAR, d', and criterion for each subject and condition. This will be used for stats as well as visualization. The second data frame, *groupDataAll*, will include every trial for every participant. We'll only use this for exporting data to use in the SDT Kamu app to check that the script works properly.

```
groupDataSDT = pd.DataFrame(columns = ['subj', 'cond', 'HR', 'FAR', 'dPrime', 'criterion'])

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

3. Now it's time to l**oop over every subject and fill up the group-level data frames**. Since this repeats much of the script from part 4, we'll look at the whole thing in context, highlighting additions needed for group-level processing.

```
rowIndex = 0
conditions = ['all', 'long', 'short']

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

    ##3B. clean up data frame

    #select only the columns we need
    rawData = pd.DataFrame(rawData, columns = ['participant',
                                               'durationCondition',
                                               'position',
                                               'key_resp_main.keys'])

    #rename those columns to more user-friendly labels
    newLabels = {
        'participant': 'subj',
        'durationCondition': 'cond',
        'position': 'side',
        'key_resp_main.keys': 'resp'
        }

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

```
    #remove rows that != trials
    isTrials = rawData.cond.notnull() #condition != NaN
    trialData = rawData[isTrials]

    #index -> trial number
    trialData = trialData.reset_index(drop=True) #reset index
    trialData.index += 1 #start from 1, not 0

    #subject name to use later
    thisSubject = trialData.loc[1, 'subj']

    #add trialData to groupDataAll
    groupDataAll = pd.concat([groupDataAll, trialData])

    ##3C. calculate SDT metrics
    #note: here the left grating will be defined as the "target"

    #condition variables
    isTarget = trialData.side == 'left'
    notTarget = trialData.side != 'left'

    respTarget = trialData.resp == 'left'

    condLong = trialData.cond == 'long'
    condShort = trialData.cond == 'short'

    #number of trials with target presented + not presented (all, long,
and short conditions)
    targetTrials = {}
    notTargetTrials = {}

    targetTrials['all'] = len(trialData[isTarget])
    notTargetTrials['all'] = len(trialData[notTarget])

    targetTrials['long'] = len(trialData[isTarget & condLong])
    notTargetTrials['long'] = len(trialData[notTarget & condLong])

    targetTrials['short'] = len(trialData[isTarget & condShort])
    notTargetTrials['short'] = len(trialData[notTarget & condShort])
```

```
    #QC that found number of trials == the expected amount
    expectedTargets = {'all': 40, 'long': 20, 'short': 20}
    expectedNotTargets = {'all': 40, 'long': 20, 'short': 20}

    for cond in conditions:
        if targetTrials[cond] != expectedTargets[cond] or notTargetTrials[cond] !=
expectedNotTargets[cond]:
            print(thisSubject, '- Possible error with trial counts')
            print('Expected targets:', expectedTargets, 'Found targets:', targetTrials)
            print('Expected non-targets:', expectedNotTargets, 'Found non-targets:',
notTargetTrials)


    #hit and FA count
    hitTrials = trialData[isTarget & respTarget]
    faTrials = trialData[notTarget & respTarget]

    hitCount = {}
    faCount = {}

    hitCount['all'] = len(hitTrials)
    faCount['all'] = len(faTrials)

    hitCount['long'] = len(hitTrials.loc[condLong])
    faCount['long']= len(faTrials.loc[condLong])

    hitCount['short'] = len(hitTrials.loc[condShort])
    faCount['short'] = len(faTrials.loc[condShort])

    #adjust to avoid infinte d'/criterion
    for cond in conditions:
        #adjust hit count
        if hitCount[cond] == targetTrials[cond]:
            hitCount[cond] -= 1
            print(thisSubject, '- Hit count for condition =', cond, 'adjusted to',
hitCount[cond])
        #adjust fa count
        if faCount[cond] == 0:
            faCount[cond] = 1
            print(thisSubject, '- FA count for condition =', cond, 'adjusted to 1')
```

```
# compute hit and FA rates
hitRate = {}
faRate = {}

for cond in conditions:
    hitRate[cond] = hitCount[cond]/targetTrials[cond]
    faRate[cond] = faCount[cond]/notTargetTrials[cond]

##3D. add d' and criterion to groupDataSDT
#[subject, condition, HR, FAR, d', criterion]

#long condition
groupDataSDT.loc[rowIndex] = [thisSubject,
                              'long',
                              hitRate['long'],
                              faRate['long'],
                              dPrime(hitRate['long'], faRate['long']),
                              criterion(hitRate['long'], faRate['long'])]
rowIndex += 1

#short condition
groupDataSDT.loc[rowIndex] = [thisSubject,
                              'short',
                              hitRate['short'],
                              faRate['short'],
                              dPrime(hitRate['short'], faRate['short']),
                              criterion(hitRate['short'], faRate['short'])]
rowIndex += 1
```

Notice that despite the loop being very, very long, almost all of it is familiar from the single-subject script, with the few additions similar to those already seen in the group-level RT script.

Before the loop, we define the variable *rowIndex* to keep track of the next row index for groupDataSDT, as well as a list of conditions to loop over.

Then the loop processes the data as in the single-subject script until we have a data frame that only includes trials. This is then added to the groupDataAll data frame.

Analysis then proceeds as described in part 4. Once this is complete, we can add HR, FAR, d', and criterion to groupDataSDT for each condition. The end result should look something like:

	subj	cond	HR	FAR	dPrime	criterion
0	S1	long	0.95	0.05	3.289707	3.330669e-16
1	S1	short	0.65	0.15	1.421754	3.255565e-01
2	S2	long	0.95	0.20	2.486475	-4.016162e-01
3	S2	short	0.75	0.10	1.956041	3.035309e-01

4. With a data frame filled with d' and criterion for every subject and condition, we can easily **compute the mean** of each for both conditions, as well as **perform a t-test** to check for statistical significance between conditions.

   We'll start by defining variables for each condition, which then allows us to select data for just that condition.

```
#variables for indexing
groupLongCond = groupDataSDT.cond == 'long'
groupShortCond = groupDataSDT.cond == 'short'

#select data
groupDataLong = groupDataSDT[groupLongCond]
groupDataShort = groupDataSDT[groupShortCond]
```

   Now all that's left to do is print out the mean d' and criterion for each condition . . .

```
print("Mean d' (long):", groupDataLong.dPrime.mean())
print("Mean d' (short):", groupDataShort.dPrime.mean())

print('Mean criterion (long):', groupDataLong.criterion.mean())
print('Mean criterion (short)', groupDataShort.criterion.mean())
```

   . . . as well as a t-test for each.

```
dPrimeStats = ttest_ind(groupDataLong.dPrime, groupDataShort.dPrime , equal_var = False)
print("T-test (d'):", dPrimeStats)

criterionStats = ttest_ind(groupDataLong.criterion, groupDataShort.criterion, equal_var = False)
print("T-test (criterion):", criterionStats)
```

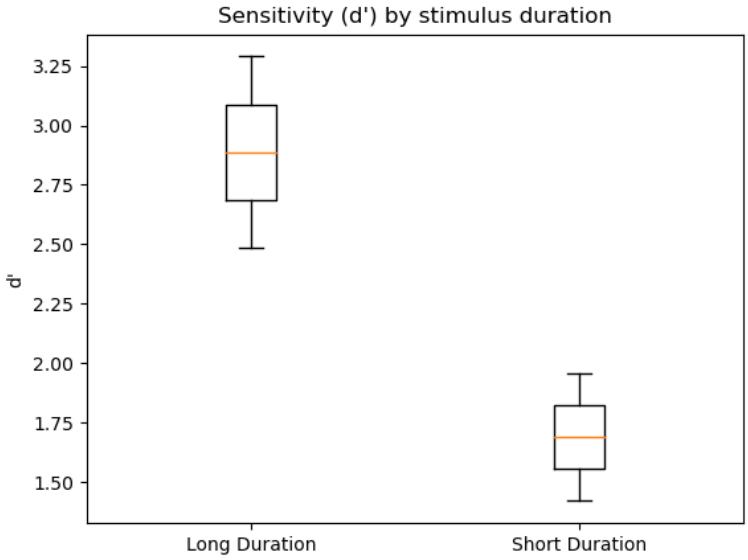
5. Once again, if we want to **format our data to check with the SDT Kamu app**, we can do so by copying groupDataAll and replacing 'left' and 'right' with 1 and 0 respectively.

```
appData = groupDataAll.copy()

appData = appData.replace({'left': 1, 'right': 0})

#save as new .csv file
pathOut = './exported_data/'
fileOut = 'export1.csv'

appData.to_csv(pathOut + fileOut, sep=';')
print('Data exported to:', pathOut + fileOut)
```

Part 6: Visualizing

Description: We can visualize d’ and criterion using the same techniques discussed for RT data. In this case, a box plot can be especially illustrative.

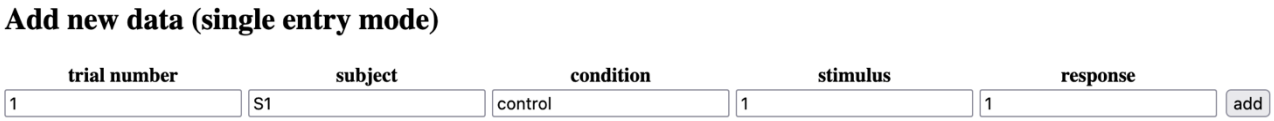
```
fig, ax = plt.subplots()

box = ax.boxplot([groupDataLong.dPrime, groupDataShort.dPrime])

ax.set_ylabel("d'")
ax.set_title("Sensitivity (d') by stimulus duration")
ax.set_xticklabels(['Long Duration', 'Short Duration'])

plt.show()
```



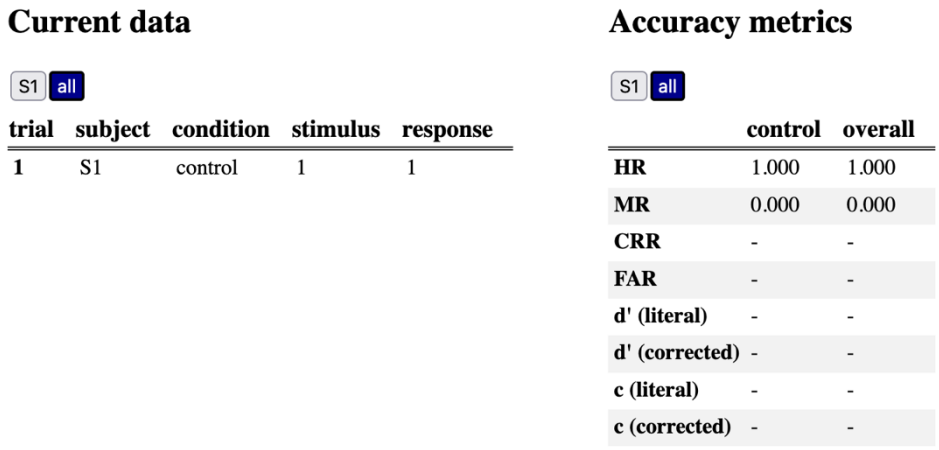
Part 7: Checking your script with SDT Kamu

Description: Once you've written a script to process SDT data for the group project, how can you be sure that it's outputting the correct results? SDT Kamu—a simple, web-based app developed for this course—is here to help, providing the tools to quickly check that you're getting the right values. You can find it at: https://psychophysics-app.utu.fi/

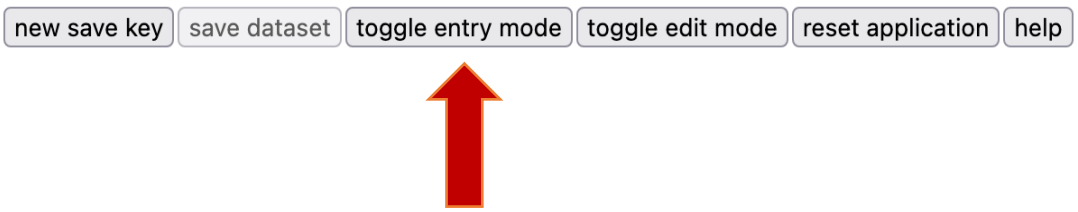
1. SDT Kamu takes trial-level data as an input. Both stimulus and response are encoded with 1s and 0s. To become familiar with the structure of the input data, you can add data trial-by-trial.



As you add trials, you'll see the input data appear below.



2. Adding data trial-by-trial is tedious, so you'll probably want to use bulk entry mode. Switch to bulk entry by clicking "toggle entry mode" in the menu bar.



Now we can enter .csv-structured data separated with semicolons, just like the data we exported at the end of parts 5 and 6. As the instructions say, don't repeat trial numbers for the same subject. This may interfere with the tables rendering properly.

Add new data (bulk entry mode)

structure each line: trialNumber;subject;condition;stimulus;response

IMPORTANT: don't repeat trial numbers!

```
71;S2;long;0;0
72;S2;short;0;0
73;S2;short;1;1
74;S2;long;0;0
75;S2;short;0;1
76;S2;short;0;0
77;S2;short;1;1
78;S2;long;0;0
79;S2;long;1;1
80;S2;long;1;1
```

Current data

S1 S2 all

trial	subject	condition	stimulus	response
1	S1	long	1	1
1	S2	long	1	1
2	S1	short	1	0
2	S2	short	0	0
3	S1	short	1	1
3	S2	long	1	1
4	S1	long	0	0
4	S2	long	1	1
5	S1	short	1	1
5	S2	long	1	1
6	S1	short	0	0
6	S2	short	1	0
7	S1	long	0	0
7	S2	long	1	1
8	S1	long	1	1
8	S2	short	0	0
9	S1	short	0	0
9	S2	long	0	0

Accuracy metrics

S1 S2 all

	long	short	overall
HR	1.000	0.650	0.825
MR	0.000	0.350	0.175
CRR	1.000	0.850	0.925
FAR	0.000	0.150	0.075
d' (literal)	Infinity	1.422	2.373
d' (corrected)	3.287	1.422	2.373
c (literal)	Infinity	0.3255	0.252
c (corrected)	0	0.3255	0.252

Notice the accuracy metrics match the output of the script in part 6 (in this case for S1). Clicking subject names allows you to toggle between them, with “all” displaying group means.

Note that the HR, MR, CRR, and FAR displayed aren’t corrected for HR = 1 or FAR = 0 (unlike the script we wrote above). However, d' (corrected) and c (corrected) use the same adjusted values as described previously.

3. SDT Kamu also gives you the option to save data sets. For instructions on this and other functionality, see “help” in the menu bar.