

# Table of Contents

<b>Downloading Psychopy</b>	<b>2</b>
<b>Components of Psychopy</b>	<b>3</b>
<b>Basics of Builder</b>	<b>4</b>
<b>How To Run “Psychopy_VMR ”</b>	<b>10</b>
<b>VMR_Builder_Code framework</b>	<b>11</b>
<b>Code Component Overview</b>	<b>13</b>
<b>Reach ‘Each Frame’ Code</b>	<b>16</b>
<b>CodeStarter</b>	<b>19</b>
<b>Pavlovia and Posting Experiments Online</b>	<b>20</b>
<b>Retrieving Data Files</b>	<b>21</b>
<b>Analysis</b>	<b>21</b>
<b>Troubleshooting</b>	<b>21</b>

# Downloading Psychopy

Installing Psychopy is simple. We are looking for the standalone package of Psychopy that will be added as an app on your laptop.

**For windows:** Go to <https://www.psychopy.org/download.html> and click on the large blue button that says “standalone package”

**For Mac:** Go to <https://github.com/psychopy/psychopy/releases> and scroll down to the “assets” tab of the most recent version of the program (right now it is 2021.1.3) and download the “Standalone...macOS.dmg”

# Components of Psychopy

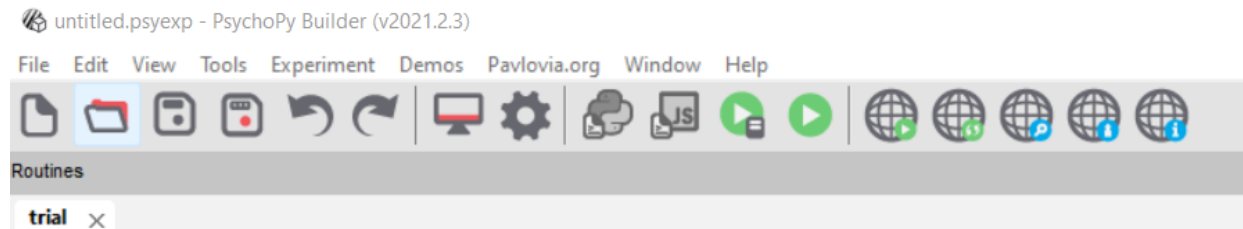
Once downloaded, the program should be able to be opened from the search bar, it has a black/white circle icon. Once opened the program will launch 3 tabs, the *runner*, the *coder*, and the *builder*.

The runner is where the output of the code from the builder or the coder will go, this tab is important to keep an eye on because it also will show errors in the code and link to where the error is actually occurring in the code, this will be a great tool for debugging. The runner can also be used as a normal 'python shell', bits of code can be put in here and run to make sure they function properly.

The coder is basically a normal python script. This can be used to write code and send it to the runner. Code that is written in the coder cannot be sent to **Pavlov** to be run online, so that is why we will be focusing on the builder view.

# Basics of Builder

Builder is where the 'coding' in Psychopy will be done, this is where the experiment will be created and run from, this in essence is what Psychopy is. It is different from 'normal' coding, although once you are able to get a handle on it, the benefits it offers are clear to see.



Above are the tabs in a Builder workspace. I will start at the top left icon and work to the right, most are self explanatory, although some are Psychopy exclusive and very important, they will be bolded.

*Create:* Creates a new builder script

*Open:* Opens a previously created builder script

*Save:* Saves current script to current name

*Save As:* Saves current script under different name

*Undo:* Undo

*Redo:* Redo

*Monitor Settings:* Used to calibrate monitor settings, original monitor settings have worked for me thus far

*Experiment Settings:* Used to change settings related to experiment settings like participant inputs, what units the experiment is in, what the background color is, etc. Useful to look at.

*Compile to Python:* Can compile the entire builder script into a 'coder' script.

*Compile to JS (JAVASCRIPT):* Compiles the entire builder script into a Javascript script

*Send to Runner:* Sends experiment to runner, but has to be run manually

**Run Experiment:** Runs experiment

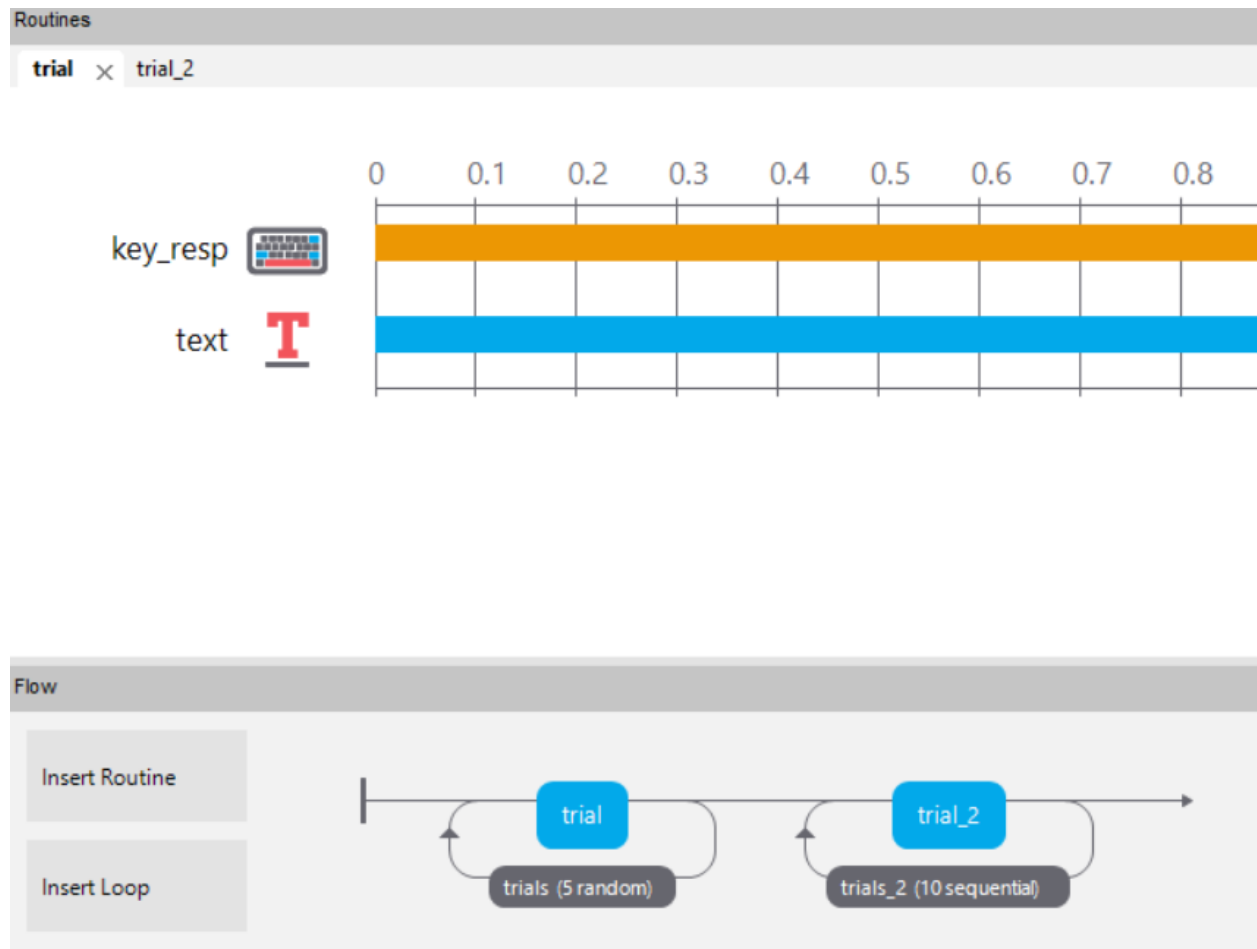
*Run Study OL:* To be used when first launching an experiment onto Pavlovia, but I would use sync instead

**SYNC:** Syncs the current experiment with Pavlovia. If there are currently no projects with your experiment name tied to your Pavlovia account, Psychopy will give you the option to create one. Must be logged in to use this feature. Will go into more detail later in the Pavlovia section

*Find Experiments OL:* Find pavlovia experiments, would be much easier to use the actual web interface of Pavlovia though

*Login to Pavlovia:* Connects Psychopy app with your existing Pavlovia account

*View:* Shows a short blurb about the current experiment



Above are the 'Routines' and 'Flow'. Builder works by sequentially stringing together what are called routines into the flow to create the framework for the overall experiment. In the image above, there are two routines created, 'trial' and 'trial\_2'.

*Routines* are where the components of the builder code will be placed, in the image above two components have been placed into 'trial' routine, a 'key response' and a 'text component'. Theoretically, this routine would have text instructions telling the participant to make a key press, then once pressed, the routine would end. In the bottom of the image we can see the flow, 'trial' is encompassed by a *loop*, and is then followed by 'trial\_2' which also has a loop. The loop dictates how many times the routine will be gone through before moving onto the next routine in the flow. The flow above shows the 'trial' routine running 5 times within the loop then moving to the 'trial\_2' routine which runs 10 times.

trials Properties ✕

Name

loopType  ▼



Is trials ☒

---

nReps \$

Selected rows

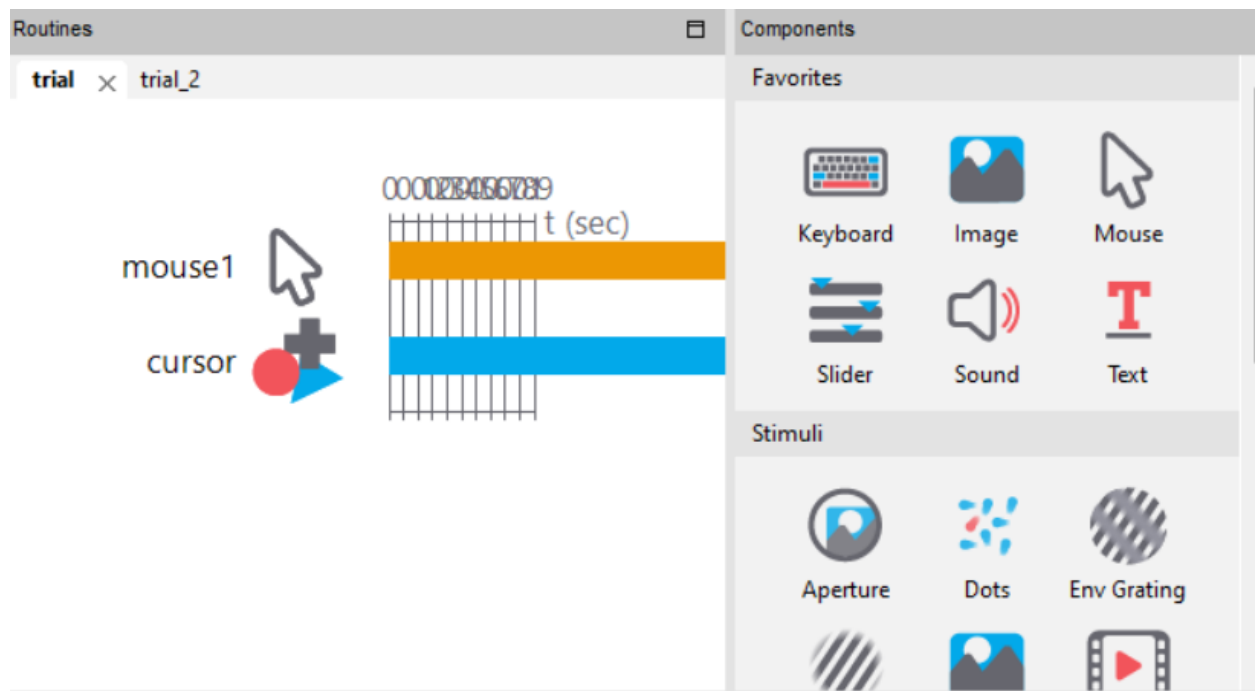
random seed \$

Conditions   

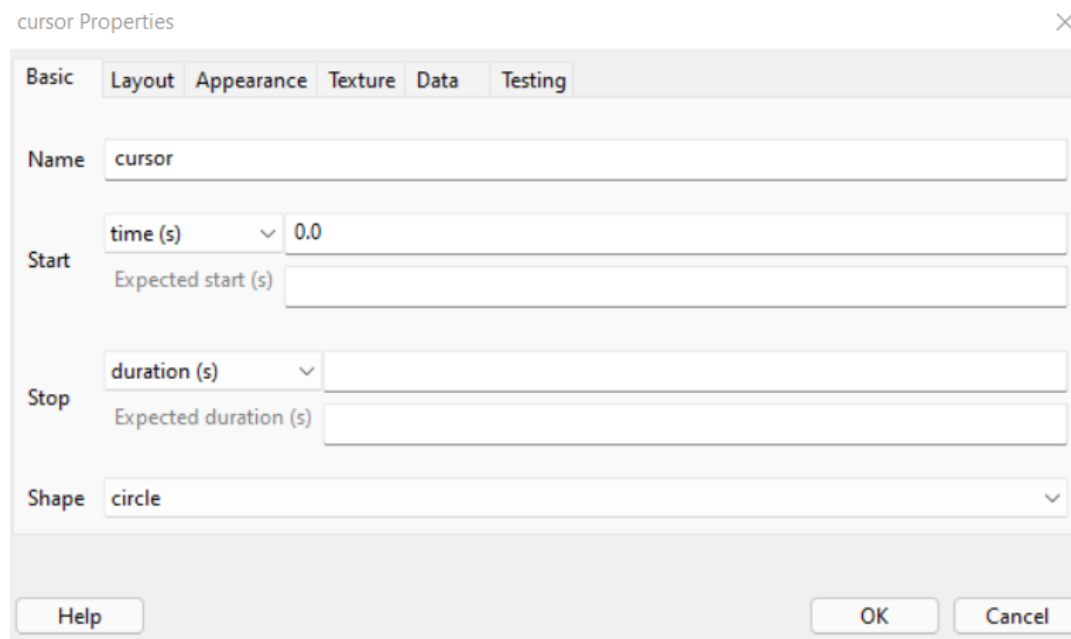
No parameters set

Help OK Cancel

Here is an example of what the 'trials' loop looks like once clicked on. It is very important that **“Is trials” is always checked** so that each time a routine runs its data can be recorded properly. 'nReps' refers to how many times the loop will go through the routine, although if using a “Conditions” file, this should probably be set to 1, or else the entire conditions file will be run that many times. A “Conditions” or “target” file (in the form of an excel sheet) can be inserted if the open folder is clicked on, this allows the experiment to be run according to certain conditions that the experimenter makes, and also this conditions file can be reference within a code component of the routine. Looptype can determine if the target file is read sequentially or random. It is also important to note that loops **should not span multiple routines**, this will mess up how the target file is read.



Above we have the components and routines sections. Components are the “things” that go into a routine, anything from a keyboard press to the mouse to the code within the routine are considered components.



Above are the tabs within the polygon component, which will be a main component that you will work with. The important tabs to keep in mind are *Basic*, *Layout*, and *Appearance*, the others

can be left how they are. Say we wanted to map a polygon as the cursor that shows up when the mouse is moved during the experiment. First we would make a cursor polygon. We would then **set 'stop' duration as nothing**, this is important to do for every component so that the component does not go away at a certain time duration, instead it can be controlled through a code component of code. We then make the cursor whatever shape we want, in this case a circle.

The screenshot shows the 'cursor Properties' dialog box with the 'Layout' tab selected. The dialog has a title bar with a close button (X) and a tabbed interface with 'Basic', 'Layout', 'Appearance', 'Texture', 'Data', and 'Testing' tabs. The 'Layout' tab contains four settings:

- Size [w,h]**: A text field with the value `(0.20, 0.20)` and a dropdown menu set to 'constant'.
- Position [x,y]**: A text field with the value `(mouse1.getPos()[0], mouse1.getPos()[1])` and a dropdown menu set to 'set every frame'.
- Spatial Units**: A dropdown menu set to 'from exp settings'.
- Orientation**: A text field with the value `0` and a dropdown menu set to 'constant'.

At the bottom of the dialog are three buttons: 'Help', 'OK', and 'Cancel'.

We then move to the *Layout* tab. The size of the cursor can be set here, in this case it is set in arbitrary units that correspond to the mouse on the screen. The position is set in a way that it reads the mouse's X and Y coordinates and maps them to the polygon. This documentation can be found [here](#), Psychopy's website has good documentation for many components. It is important to make sure it says **'set every frame'** next to the position, this way the cursor maps to the mouse every frame and does not stay still. The *Appearance* tab can be used to change the color and opacity of the polygon being used.



# How To Run “Psychopy\_VMR ”

In order to run the code locally, you will need to 1) have the Psychopy client installed 2) have downloaded the “VMR\_Builder\_code.psyexp” builder code and the “target\_file\_real.xlsx” excel file from github and 3) placed those two files within the same local folder.

Link to [Github repository](#)

To download the required materials from Github, click the green ‘code’ button, then *Download Zip*. This will download a zip folder containing the contents of the github repository, which you will then have to unzip.

Once the client is running, *open* the “VMR\_Builder\_code.psyexp” file within the builder. The builder file should open and show a Flow at the bottom that goes “Instr→code\_starter→Reach...” and the routines should be displayed at the top. To run the code, click the green arrow button *Run Experiment*. The client will ask for a ‘participant’ and ‘session’, but you are able to just click “OK” without inputting any information. The task will start and pull the target values from the target file, as long as they are in the same local folder. The task has 128 reaches built in, but you can click “ESC” at any time to stop the task. After the program is done, the data will be saved to a new folder called ‘data’ that holds data related to the run.

# VMR\_Builder\_Code framework

The overall framework of the code is as follows:

1. Instructions followed by 16 loops of the *Reach* routine (OLFB and EPFB)
2. Instructions followed by 80 loops of the *Reach\_VMR* routine
3. Instructions followed by 16 loops of the *Reach* routine
4. Instructions followed by 16 loops of the *Reach* routine (OLFB and EPFB)

trials Properties

Name: trials

loopType: sequential

Is trials: ☒

nReps: 1

Selected rows: 1:16

random seed: \$

Conditions: Target\_file\_real.xlsx

128 conditions, with 8 parameters [trialnum, tgt\_distance, tgt\_angle, rotation, online\_fb, endpoint\_feedback, VMR, between\_blocks]

Help OK Cancel

Above is the *loop* for the first set of reaches. The 'selected rows' are rows 1 through 16 in the target file, this will read those rows and take the information from the columns. The first set of reaches is for the participant to get settled into how the reaching paradigm works without any rotation applied. This set of reaches has both Online Feedback (OLFB) and EndPoint Feedback (EPFB).

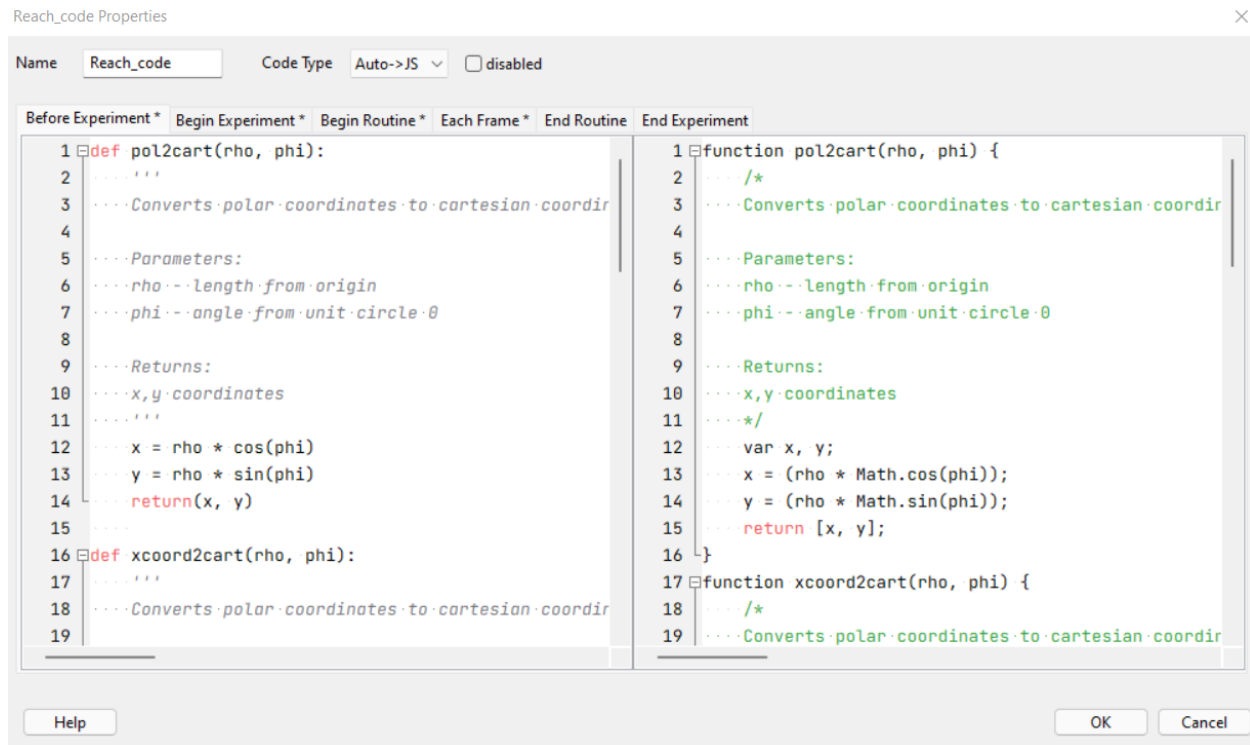
The participant then gets instructions and presses enter for the next phase to move on. They then make 80 reaches in the *Reach\_VMR* loop, the cursor polygon is rotated 30 degrees CCW from the actual mouse position.

The participant then gets more instructions and is placed in the *washout\_nofb* loop, this loop uses the same *Reach* routine as before but has different 'selected rows' in the target file, these rows have the OLFB and EPFB turned off. The participant makes reaches without knowing where the cursor is.

The participant then gets more instructions and is placed in the *washout\_fb* loop; this loop uses the same *Reach* routine as before but has different ‘selected rows’ in the target file. The participant makes reaches similar to the first set of reaches.

## Code Component Overview

The ‘code’ component is the core of how you will program what happens within your experiment. I will be using a code component that I created within the *VMR\_Builder\_Code* task to display how the code is organized.



Above is the screen that opens when a code component is created. At the top left is the name of the component and to the right is the ‘Code Type’. The Code Type describes how the python code on the left of the screen (That you will type) will be converted. Right now it is set to Auto->JS, which stands for “automatically convert to Javascript”, this means that the code you write on the left is directly converted to JS on the right side of the screen. The Psychopy app on your computer will run the code in Python, but once the code is posted online, Pavlovia will read the code in its Javascript form, meaning that both sections of the code must work properly for you to post it online.

There are six tabs of code within the code component, *Before Experiment*, *Begin Experiment*, *Begin Routine*, *Each Frame*, *End Routine*, and *End Experiment*. Making sure that certain code is in the right place is essential for the overall routine to run properly, and this should be the **first thing you check when troubleshooting**.

*Before Experiment:* The code that goes in this tab are things (mainly functions) that will be used throughout the experiment. As shown above, this is where I placed my essential functions like 'pol2cart'. Throughout my experiment, I have one code component per routine, these code components each have all six tabs but are usually only needed for the 'begin routine' and 'each frame' tabs, so it is recommended that all functions be placed in **one 'Before Experiment' tab** so it is easy to keep track of. The functions placed in *Before Experiment* tab will be able to be used in any code component throughout experiment.

*Begin Experiment:* The code in the *Begin Experiment* tab runs after the *Before* tab but before the *Begin Routine* tab. This is where global variables should be initialized. For example, if a variable is needed to set the position of a polygon in a routine you will need to initialize the polygon like "polygonx = 0, polygony = 0" within this tab so that polygonx and polygony can have a value before the routine begins, otherwise there will be an error (if there is not an error in Python there will probably be an error once posted online)

*Begin Routine:* The code in this tab will begin running corresponding to the routine it is placed in. Local variables corresponding to the current routine should be placed in this tab. As an example: If you need a 'counter' variable you would be able to initialize this variable within this tab as 'counter = 0', you could then manipulate the counter within the *Each Frame* tab to make use of it. Then the routine will end and (given there is a loop running) the loop will bring the routine back to the *Begin Routine* tab where the counter will be initialized as 0 again. This tab would be where you set the actual value of polygonx and polygony that were initialized above

```
16 #Convert target file to radians
17 rad1 = radians(tgt_angle)
18
19 #Set the target's x,y position
20 Targetxx = xcoord2cart(tgt_distance, rad1)
21 Targetyy = ycoord2cart(tgt_distance, rad1)
22
```

Above is a snippet of my code from the *Begin Routine* tab. Targetxx and Targetyy were initialized as 0 in the *Begin Experiment* tab, but are now being set as values within the *Begin Routine*. The variables "tgt\_angle" and "tgt\_distance" correspond to variables that are embedded within the target file, and "x/ycoord2cart" are functions that were within the *Before Experiment* tab. This code snippet will take the angle and distance values within my target file and apply them to the Targetxx and Targetyy variables, which are connected to a polygon component named target, this polygon will then be in a certain position when the routine begins based on these values.

*Each Frame*: This is the code tab that will contain the meat of your code. The *Each Frame* tab runs like its name, every frame, so this is the primary measurement of time within the Psychopy code. The code within this tab will vary wildly depending on the experiment, although it would be a good idea to section this tab into 'phases'.

Before Experiment *	Begin Experiment *	Begin Routine *	Each Frame *	End Routine
<pre> 11 if phase == 0: 12     ... Invis_Home.opacity = 1; 13     ... target.opacity = 0; 14     ... 15     ... if Dist_from_Home &gt; 0.15: 16         ... continueRoutine = True; 17         ... Cursor_2.opacity = 0 18     ... if (Dist_from_Home &lt;= 0.15) &amp; (Dist_from_Home &gt; 19         ... continueRoutine = True; 20         ... 21         ... Cursor_2.opacity = 1 22     ... if Dist_from_Home &lt; 0.0125: 23         ... wait_time = wait_time+1 24     ... if wait_time &gt; 20: 25         ... phase=1; 26     ##### 27     #Phase 1 28     #REACH Phase </pre>				

Above is a section of code from my *Each Frame* tab. This code is made to help the user find the home position with their cursor, changing the opacity of the cursor as it gets closer to home. The 'if' statement at the top determines what phase the code is currently in, right now the phase is 0 (phase variable was initialized in the *Begin Routine* tab). The code is run every frame, but the phase variable keeps the code within just this top section, once a certain threshold is reached, then the phase changes and the code moves on to the next phase.

*End Routine*: This code runs after the *Each Frame* tab but before the routine actually ends. This would be the place where variables could be set to certain values or where an output could be sent to the runner to see if the code is working correctly. Extra data could also be collected here. I did not find much use for this tab.

*End Experiment*: This code runs at the very end of the experiment. I did not find a use case for this tab.

## Reach 'Each Frame' Code

This section will be dedicated to going through the *Reach* code, specifically the *Each Frame* tab. The other tabs are, in general, initializing variables that appear within this section of code.

```
1 #Find distance from origin to cursor at every frame
2 Dist_from_Home = sqrt(((Cursor_2.pos[0]-Invis_Home.pos[0])**2)+(C
3
4 #####
5 #Phase 0, using 'counter' to change phases
6 #Finding Home phase
7 #When cursor within certain distance of home, becomes visible
8 #When its within home radius, waits 20 frames then goes to next pt
9
10
11 if phase == 0:
12     Invis_Home.opacity = 1;
13     target.opacity = 0;
14     ...
15     if Dist_from_Home > 0.15:
16         continueRoutine = True;
17         Cursor_2.opacity = 0
18     if (Dist_from_Home <= 0.15) & (Dist_from_Home > 0.0125):
19         continueRoutine = True;
20         ...
21         Cursor_2.opacity = 1
22     if Dist_from_Home < 0.0125:
23         wait_time = wait_time+1
24     if wait_time > 20:
25         phase=1;
```

The first variable in the code is *Dist\_from\_Home*, this is not within any 'game phase' and therefore is record every frame. It measures how far the mouse is from the center of the screen (called home) at every frame. This distanced is used in the other phases in the game.

We start in "Phase 0" where the variable phase equals 0. The home opacity starts at 1 (you can see it) and the targets start at 0 (they are invisible). The cursor is somewhere on the screen, but it is invisible if it is too far away from the home position, the user must move it around to find the

center of the screen. The code is then checking to see what the `dist_from_home` is, if it is greater than 0.15, then the cursor stays invisible. If it gets closer and is between 0.15 and 0.0125 then the cursor becomes visible. If the cursor is within 0.0125 (the home 'circle') then the code adds to the `wait_time` variable for 20 frames and the phase equals 1.

```
27 #Phase 1
28 #REACH Phase
29 #Checks target file for online feedback y/n
30 #Once cursor crosses 'target ring', goes to next phase
31
32 if phase == 1:
33     if online_fb == 1:
34         Cursor_2.opacity = 1;
35     if online_fb == 0:
36         Cursor_2.opacity = 0;
37     Invis_Home.opacity = 0.0;
38     target.opacity = 1;
39
40
41 if Dist_from_Home > 0.354:
42     Cursor_2.fillColor = 'gray';
43     Cursor_2.opacity = 0;
44     phase = 2
45
```

Phase 1 is the 'Reaching for target' phase. The variable `online_fb` comes from the target file, if it is 1 then the cursor becomes visible, if 0 then invisible. The home icon also becomes invisible while the target becomes visible. The participant will then reach for the target and once the cursor crosses the 'target ring threshold' (eg. reaches to the ring that the target is placed on) the cursor becomes invisible and the phase changes to 2.

```

46 #Phase 2
47 #Endpoint phase
48 #Checks target file for endpoint y/n
49 #Tx calculates the phi of the cursor
50 #Tx plugged into the endpointmarker
51 #endpoint stays for 20 frames, then routine ends
52
53
54 if phase == 2:
55     if endpoint_feedback == 1:
56         EndpointMarker.opacity = 1;
57     if endpoint_feedback == 0:
58         EndppointMarker.opacity=0;
59     #Cursor_2.opacity=1;
60     .....
61     Tx = JustPhi((Cursor_2.pos[0]-Invis_Home.pos[0]),(Cursor_2
62     wait_time2 = wait_time2+1
63     .....
64     if wait_time2 == 1:
65         EndPointCursorx = xcoord2cart(0.354,Tx);
66         EndPointCursory = ycoord2cart(0.354,Tx);
67     if wait_time2 > 20:
68         continueRoutine = False;
69 else:
70     continueRoutine = True;

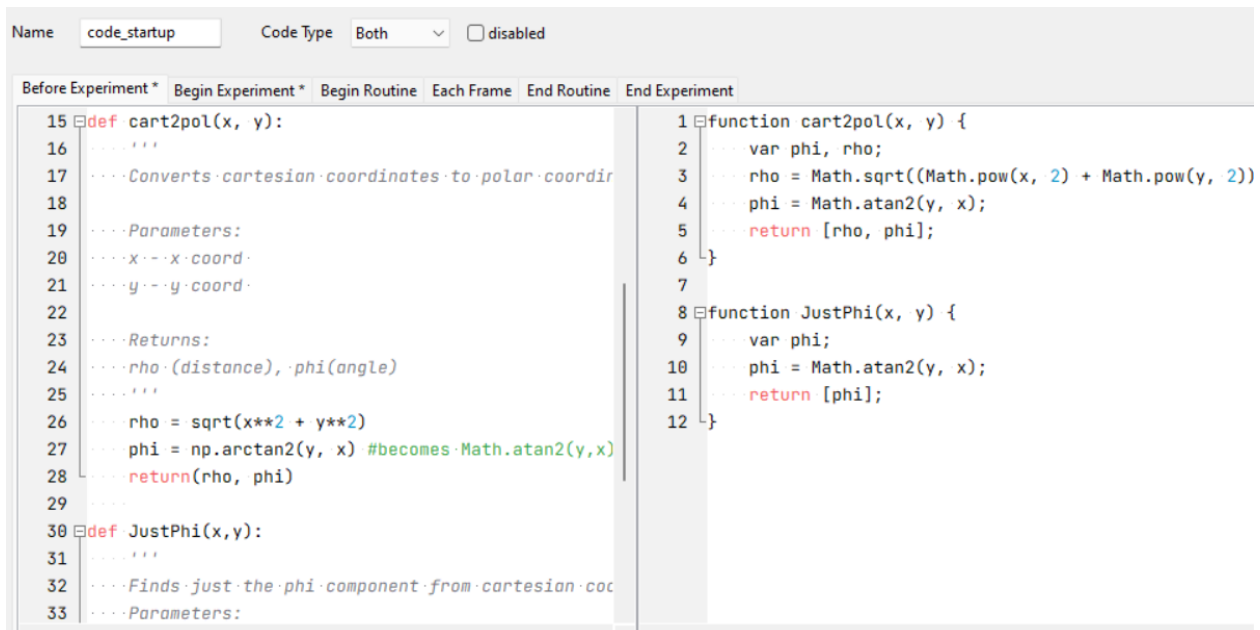
```

Like `online_fb`, `endpoint_feedback` is from the target file, the code checks if the target file wants there to be endpoint feedback or not (these variables are useful because it allows use to use the same *Reach* routine multiple times, only having to change the what's in the target file). The variable `Tx` is used to calculate the angle of the endpoint marker (which is a different polygon from the cursor) using the *JustPhi* function from the *codestarter* routine. Then the `Endpointcursorx` and `y` are set to the coordinates of where the mouse crossed the target ring. After 20 frames the routine ends.



# CodeStarter

Psychopy is very user friendly, especially if you have some knowledge of Python. There are however some quirks that make Psychopy confusing to use. Almost every piece of code can be nicely converted from Python to Javascript (JS is what is used when the code is posted online) using the Auto→JS tab at the top. Psychopy does not allow the use of 'Python libraries' to be converted to JS point in time, this includes things like *numpy* or *math* python libraries, so instead this section of code uses **Code Type: Both**.



The screenshot shows the Psychopy CodeStarter interface. At the top, there is a 'Name' field with 'code\_startup' and a 'Code Type' dropdown set to 'Both'. Below this is a row of tabs: 'Before Experiment \*', 'Begin Experiment \*', 'Begin Routine', 'Each Frame', 'End Routine', and 'End Experiment'. The 'Begin Routine' tab is selected. The interface is split into two panes. The left pane contains Python code for two functions: `cart2pol(x, y)` and `JustPhi(x, y)`. The right pane contains the JavaScript equivalents of these functions. The Python code uses `np.arctan2` and `sqrt` from the `numpy` library. The JavaScript code uses `Math.sqrt` and `Math.atan2`. The code is as follows:

```
15 def cart2pol(x, y):
16     """
17     ...Converts cartesian coordinates to polar coordinates
18     ...Parameters:
19     ...x--x coord
20     ...y--y coord
21     ...Returns:
22     ...rho (distance), phi (angle)
23     """
24     rho = sqrt(x**2 + y**2)
25     phi = np.arctan2(y, x) #becomes Math.atan2(y,x)
26     return(rho, phi)
27
28 def JustPhi(x,y):
29     """
30     ...Finds just the phi component from cartesian coordinates
31     ...Parameters:
```

```
1 function cart2pol(x, y) {
2     var phi, rho;
3     rho = Math.sqrt((Math.pow(x, 2) + Math.pow(y, 2)))
4     phi = Math.atan2(y, x);
5     return [rho, phi];
6 }
7
8 function JustPhi(x, y) {
9     var phi;
10    phi = Math.atan2(y, x);
11    return [phi];
12 }
```

The functions in the code starter use the python library `numpy` or 'np' to use the `arctan` function. This `np.arctan2` does not convert to JS (it will convert, but give an error when running in Pavlovia). So in this section of code I manually wrote both the left and right sides. The syntax for the left and right sides are slightly different and you may need to consult a JS guide to write the JS code if this is needed for another function. I would be very careful when changing either side of the code because unlike the other sections of code which tell you 'syntax error' on the right side if the python code is not written correctly, this does not do that.

# Pavlovia and Posting Experiments Online

Once an experiment is working within the Python runner, it's time to post on Pavlovia!

Pavlovia homepage link: <https://pavlovia.org/#main>

Pavlovia is a hub for people around the world to post and host online experiments. At this point in time the site is mainly used for behavioral experiments, although more motor experiments will hopefully come to the site soon. In order to explore the tabs on the site you must first register and login. The *Explore* tab lets you browse other public experiments, you can play them in the browser and also download their corresponding git files so that you can look at the builder code in your Psychopy app. The *Dashboard* is where your profile and experiments are housed, this is where once posted, you can change settings related to your online experiment.

## Steps to post

1. Login to Pavlovia in your Psychopy app (Globe icon with blue person)
2. Click the Sync with Web Project button (Globe icon with green swirl)
  - a. If this is your first time posting your project you will get a popup window asking you to name your project, title your project the same name as it is titled in your folder.
3. A popup window will ask you to title your 'changes', this comes up everytime you sync your project with Pavlovia, this will keep a log in github of the changes you made so you are able to keep track.
4. The runner will then prompt you of your successful sync, usually this takes less than 10 seconds
5. You can then navigate to your Dashboard on Pavlovia in the browser and go to the experiments tab. Here you will see the experiment you just posted.
6. The experiment will be set to 'inactive' but you can change it to 'pilot' which allows you to demo your experiment in the browser to make sure that the Javascript works correctly, or set it to 'running' which allows you to get a shareable link and collect data from others. In order for the experiment to be set to running there must be enough credits allocated to the experiment.

# Retrieving Data Files

## How Data is Saved:

mouse\_2 Properties ×

Basic Data Testing

Save mouse state final ▼

Time relative to mouse onset ▼

Store params for clicked \$ name,

Save onset/offset times ☒

Sync timing with screen refresh ☒

Above is an image of the properties of a *mouse* component. For this experiment, the measure of importance is the hand angle difference, or the difference between the target and the position of the mouse in degrees. We already know the target's position from the target file. The mouse position can be found in the 'data' file, in cartesian coordinates (which will need to be converted to degrees). In order to save the mouse state at the end of the routine, it is important to **check Save Mouse State: Final**.

## Local Data Files:

Local data files will be placed in the folder titled 'data', this folder is made whenever you run the code for the first time. The data is outputted as PSYDAT, Text, and Excel documents, but I have found the most use for the Excel files.

## Online Data Files:

Once you have started piloting/running the experiment on Pavlovia, the 'Download Results' tab will show up on the Experiment's page. Click download results and the Data will be downloaded as a zip file with the same contents as the Local Data folder.

# Analysis

The data analysis template is in the github folder under the name 'HandAngleAnalysis.ipynb'. This is a python file that can be opened and run in a jupyter notebook. The is a template for analyzing the data that has been retrieved above, the current lab member's data is still in the template to show how the analysis is going to run. I manually inputted data into the notebook, but this can be done much easier with a data frame and the *load\_workbook()* function, although you might have to mess around with the data that is output from Psychopy.

# Troubleshooting

There were three main problems I ran into while creating and posting this experiment.

## JavaScript Conversions:

In the 'codestarter' section of this guide, I went through the importance of keeping an eye on how things are converted from Python to JS. In cases where conversions are not happening automatically, or the conventions do not work in Pavlovia, it may be worth a shot looking at the [Python to JavaScript crib sheet](#). This is a document that details the dos and don'ts of converting from Python to JS. There are some minor conventional things that you do not notice as a problem (because the Python runner is working fine) but that will halt the experiment from running online. The main thing to think about is posting the experiment and piloting early and often, so you can make sure that the python and JS are working properly.

## JavaScript: No python libraries at the moment

Using python libraries is a no-brainer when it comes to crafting experiments online, they are extremely useful and sometimes essential (the only way to use arctan is through the numpy library). It is important to keep in mind that when using a python library, it will work in the Psychopy runner but not on Pavlovia, unless you manually find out how to write the code in JS. Once you understand this it is easier to be conscientious of the code you are write to either exclude libraries or manually translate. This may also be obsolete in newer versions of the software.

## Posting Online

Sometimes the experiment will give an error online that it did not give in the Psychopy runner. When this happens, open the console in the browser. In Chrome, use **Ctrl Shift J (on Windows) or Ctrl Option J (on Mac)**. This will open the console on the right side of the screen. You will be able to navigate to the source of the error