
PsychoPy - Psychology software for Python

Release 1.84.0rc5

Jonathan Peirce

July 17, 2016

1	About PsychoPy	1
2	General issues	3
3	Installation	29
4	Dependencies	33
5	Getting Started	35
6	Builder	41
7	Coder	69
8	Reference Manual (API)	83
9	Troubleshooting	143
10	Recipes (“How-to”s)	147
11	Frequently Asked Questions (FAQs)	159
12	Resources (e.g. for teaching)	161
13	For Developers	163
14	PsychoPy Experiment file format (.psyexp)	177
	Python Module Index	181

About PsychoPy

1.1 Citing PsychoPy

If you use this software, please cite one of the papers that describe it.

1. Peirce, JW (2007) PsychoPy - Psychophysics software in Python. *J Neurosci Methods*, 162(1-2):8-13
2. Peirce JW (2009) Generating stimuli for neuroscience using PsychoPy. *Front. Neuroinform.* 2:10. doi:10.3389/neuro.11.010.2008

Citing these papers gives the reviewer/reader of your study information about how the system works, it also attributes some credit for its original creation, and it means provides a way to justify the continued development of the package.

General issues

These are issues that users should be aware of, whether they are using Builder or Coder views.

2.1 Monitor Center

PsychoPy provides a simple and intuitive way for you to calibrate your monitor and provide other information about it and then import that information into your experiment.

Information is inserted in the Monitor Center (Tools menu), which allows you to store information about multiple monitors and keep track of multiple calibrations for the same monitor.

For experiments written in the Builder view, you can then import this information by simply specifying the name of the monitor that you wish to use in the *Experiment settings* dialog. For experiments created as scripts you can retrieve the information when creating the Window by simply naming the monitor that you created in Monitor Center. e.g.:

```
from psychopy import visual
win = visual.Window([1024, 768], mon='SonyG500')
```

Of course, the name of the monitor in the script needs to match perfectly the name given in the Monitor Center.

2.1.1 Real world units

One of the particular features of PsychoPy is that you can specify the size and location of stimuli in units that are independent of your particular setup, such as degrees of visual angle (see *Units for the window and stimuli*). In order for this to be possible you need to inform PsychoPy of some characteristics of your monitor. Your choice of units determines the information you need to provide:

Units	Requires
'norm' (normalised to width/height)	n/a
'pix' (pixels)	Screen width in pixels
'cm' (centimeters on the screen)	Screen width in pixels and screen width in cm
'deg' (degrees of visual angle)	Screen width (pixels), screen width (cm) and distance (cm)

2.1.2 Calibrating your monitor

PsychoPy can also store and use information about the gamma correction required for your monitor. If you have a Spectrascan PR650 (other devices will hopefully be added) you can perform an automated calibration in which PsychoPy will measure the necessary gamma value to be applied to your monitor. Alternatively this can be added

manually into the grid to the right of the Monitor Center. To run a calibration, connect the PR650 via the serial port and, immediately after turning it on press the *Find PR650* button in the Monitor Center.

Note that, if you don't have a photometer to hand then there is a method for determining the necessary gamma value psychophysically included in PsychoPy (see `gammaMotionNull` and `gammaMotionAnalysis` in the demos menu).

The two additional tables in the Calibration box of the Monitor Center provide conversion from *DKL* and *LMS* colour spaces to *RGB*.

2.2 Units for the window and stimuli

One of the key advantages of PsychoPy over many other experiment-building software packages is that stimuli can be described in a wide variety of real-world, device-independent units. In most other systems you provide the stimuli at a fixed size and location in pixels, or percentage of the screen, and then have to calculate how many cm or degrees of visual angle that was.

In PsychoPy, after providing information about your monitor, via the [Monitor Center](#), you can simply specify your stimulus in the unit of your choice and allow PsychoPy to calculate the appropriate pixel size for you.

Your choice of unit depends on the circumstances. For conducting demos, the two normalised units ('norm' and 'height') are often handy because the stimulus scales naturally with the window size. For running an experiment it's usually best to use something like 'cm' or 'deg' so that the stimulus is a fixed size irrespective of the monitor/window.

For all units, the centre of the screen is represented by coordinates (0,0), negative values mean down/left, positive values mean up/right.

2.2.1 Height units

With 'height' units everything is specified relative to the height of the window (note the window, not the screen). As a result, the dimensions of a screen with standard 4:3 aspect ratio will range (-0.6667,-0.5) in the bottom left to (+0.6667,+0.5) in the top right. For a standard widescreen (16:10 aspect ratio) the bottom left of the screen is (-0.8,-0.5) and top-right is (+0.8,+0.5). This type of unit can be useful in that it scales with window size, unlike *Degrees of visual angle* or *Centimeters on screen*, but stimuli remain square, unlike *Normalised units*. Obviously it has the disadvantage that the location of the right and left edges of the screen have to be determined from a knowledge of the screen dimensions. (These can be determined at any point by the *Window.size* attribute.)

Spatial frequency: cycles **per stimulus** (so will scale with the size of the stimulus).

Requires : No monitor information

2.2.2 Normalised units

In normalised ('norm') units the window ranges in both x and y from -1 to +1. That is, the top right of the window has coordinates (1,1), the bottom left is (-1,-1). Note that, in this scheme, setting the height of the stimulus to be 1.0, will make it half the height of the window, not the full height (because the window has a total height of 1:-1 = 2!). Also note that specifying the width and height to be equal will not result in a square stimulus if your window is not square - the image will have the same aspect ratio as your window. e.g. on a 1024x768 window the size=(0.75,1) will be square.

Spatial frequency: cycles **per stimulus** (so will scale with the size of the stimulus).

Requires : No monitor information

2.2.3 Centimeters on screen

Set the size and location of the stimulus in centimeters on the screen.

Spatial frequency: cycles per cm

Requires : information about the screen width in cm and size in pixels

Assumes : pixels are square. Can be verified by drawing a stimulus with matching width and height and verifying that it is in fact square. For a *CRT* this can be controlled by setting the size of the viewable screen (settings on the monitor itself).

2.2.4 Degrees of visual angle

Use degrees of visual angle to set the size and location of the stimulus. This is, of course, dependent on the distance that the participant sits from the screen as well as the screen itself, so make sure that this is controlled, and remember to change the setting in *Monitor Center* if the viewing distance changes.

Spatial frequency: cycles per degree

Requires : information about the screen width in cm and pixels and the viewing distance in cm

There are actually three variants: 'deg', 'degFlat', and 'degFlatPos'

'deg' : Most people using degrees of visual angle choose to make the assumption that a degree of visual angle spans the same number of pixels at all parts of the screen. This isn't actually true for standard flat screens - a degree of visual angle at the edge of the screen spans more pixels because it is further from the eye. For moderate eccentricities the error is small (a 0.2% error in size calculation at 3 deg eccentricity) but grows as stimuli are placed further from the centre of the screen (a 2% error at 10 deg). For most studies this form of calculation is preferred, as it does not result in a warped appearance of visual stimuli, but if you need greater precision at far eccentricities then choose one of the alternatives below.

'degFlatPos' : This accounts for flat screens in calculating position coordinates of visual stimuli but leaves size and spatial frequency uncorrected. This means that an evenly spaced grid of visual stimuli will appear warped in position but will

'degFlat' : This corrects the calculations of degrees for flatness of the screen for each vertex of your stimuli. Square stimuli in the periphery will, therefore, become more spaced apart but they will also get larger and rhomboid in the pixels that they occupy.

2.2.5 Pixels on screen

You can also specify the size and location of your stimulus in pixels. Obviously this has the disadvantage that sizes are specific to your monitor (because all monitors differ in pixel size).

Spatial frequency: `'cycles per pixel'` (this catches people out but is used to be in keeping with the other units. If using pixels as your units you probably want a spatial frequency in the range 0.2-0.001 (i.e. from 1 cycle every 5 pixels to one every 100 pixels).

Requires : information about the size of the screen (not window) in pixels, although this can often be deduce from the operating system if it has been set correctly there.

Assumes: nothing

2.3 Color spaces

The color of stimuli can be specified when creating a stimulus and when using `setColor()` in a variety of ways. There are three basic color spaces that PsychoPy can use, RGB, DKL and LMS but colors can also be specified by a name (e.g. 'DarkSalmon') or by a hexadecimal string (e.g. '#00FF00').

examples:

```
stim = visual.GratingStim(win, color=[1,-1,-1], colorSpace='rgb') #will be red
stim.setColor('Firebrick') #one of the web/X11 color names
stim.setColor('#FFFAF0') #an off-white
stim.setColor([0,90,1], colorSpace='dkl') #modulate along S-cone axis in isoluminant plane
stim.setColor([1,0,0], colorSpace='lms') #modulate only on the L cone
stim.setColor([1,1,1], colorSpace='rgb') #all guns to max
stim.setColor([1,0,0]) #this is ambiguous - you need to specify a color space
```

2.3.1 Colors by name

Any of the [web/X11 color names](#) can be used to specify a color. These are then converted into RGB space by PsychoPy.

These are not case sensitive, but should not include any spaces.

2.3.2 Colors by hex value

This is really just another way of specifying the r,g,b values of a color, where each gun's value is given by two hexadecimal characters. For some examples see [this chart](#). To use these in PsychoPy they should be formatted as a string, beginning with # and with no spaces. (NB on a British Mac keyboard the # key is hidden - you need to press Alt-3)

2.3.3 RGB color space

This is the simplest color space, in which colors are represented by a triplet of values that specify the red green and blue intensities. These three values each range between -1 and 1.

Examples:

- [1,1,1] is white
- [0,0,0] is grey
- [-1,-1,-1] is black
- [1.0,-1,-1] is red
- [1.0,0.6,0.6] is pink

The reason that these colors are expressed ranging between 1 and -1 (rather than 0:1 or 0:255) is that many experiments, particularly in visual science where PsychoPy has its roots, express colors as deviations from a grey screen. Under that scheme a value of -1 is the maximum decrement from grey and +1 is the maximum increment above grey.

Note that PsychoPy will use your monitor calibration to linearize this for each gun. E.g., 0 will be halfway between the minimum luminance and maximum luminance for each gun, if your monitor gammaGrid is set correctly.

2.3.4 HSV color space

Another way to specify colors is in terms of their Hue, Saturation and ‘Value’ (HSV). For a description of the color space see the [Wikipedia HSV entry](#). The Hue in this case is specified in degrees, the saturation ranging 0:1 and the ‘value’ also ranging 0:1.

Examples:

- [0,1,1] is red
- [0,0.5,1] is pink
- [90,1,1] is cyan
- [anything, 0, 1] is white
- [anything, 0, 0.5] is grey
- [anything, anything,0] is black

Note that colors specified in this space (like in RGB space) are not going to be the same another monitor; they are device-specific. They simply specify the intensity of the 3 primaries of your monitor, but these differ between monitors. As with the RGB space gamma correction is automatically applied if available.

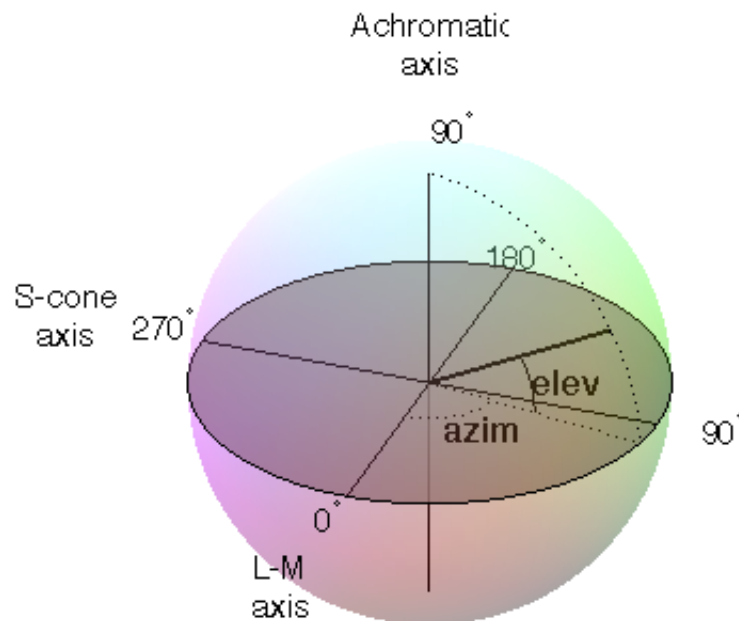
2.3.5 DKL color space

To use DKL color space the monitor should be calibrated with an appropriate spectrophotometer, such as a PR650.

In the Derrington, Krauskopf and Lennie ¹ color space (based on the Macleod and Boynton ² chromaticity diagram) colors are represented in a 3-dimensional space using spherical coordinates that specify the *elevation* from the isoluminant plane, the *azimuth* (the hue) and the contrast (as a fraction of the maximal modulations along the cardinal axes of the space).

¹ Derrington, A.M., Krauskopf, J., & Lennie, P. (1984). Chromatic Mechanisms in Lateral Geniculate Nucleus of Macaque. *Journal of Physiology*, 357, 241-265.

² MacLeod, D. I. A. & Boynton, R. M. (1979). Chromaticity diagram showing cone excitation by stimuli of equal luminance. *Journal of the Optical Society of America*, 69(8), 1183-1186.



In PsychoPy these values are specified in units of degrees for elevation and azimuth and as a float (ranging -1:1) for the contrast.

Note that not all colors that can be specified in DKL color space can be reproduced on a monitor. [Here](#) is a movie plotting in DKL space (showing *cartesian* coordinates, not spherical coordinates) the gamut of colors available on an example CRT monitor.

Examples:

- [90,0,1] is white (maximum elevation aligns the color with the luminance axis)
- [0,0,1] is an isoluminant stimulus, with azimuth 0 (S-axis)
- [0,45,1] is an isoluminant stimulus, with an oblique azimuth

2.3.6 LMS color space

To use LMS color space the monitor should be calibrated with an appropriate spectrophotometer, such as a PR650.

In this color space you can specify the relative strength of stimulation desired for each cone independently, each with a value from -1:1. This is particularly useful for experiments that need to generate cone isolating stimuli (for which modulation is only affecting a single cone type).

2.4 Preferences

The Preferences dialog allows to adjust general settings for different parts of PsychoPy. The preferences settings are saved in the configuration file *userPrefs.cfg*. The labels in brackets for the different options below represent the abbreviations used in the *userPrefs.cfg* file.

In rare cases, you might want to adjust the preferences on a per-experiment basis. See the API reference for the Preferences class [here](#).

2.4.1 Application settings (App)

These settings are common to all components of the application (Coder and Builder etc)

show start-up tips (showStartupTips): Display tips when starting PsychoPy.

large icons (largeIcons): Do you want large icons (on some versions of wx on OS X this has no effect)?

default view (defaultView): Determines which view(s) open when the PsychoPy app starts up. Default is 'last', which fetches the same views as were open when PsychoPy last closed.

reset preferences (resetPrefs): Reset preferences to defaults on next restart of PsychoPy.

auto-save prefs (autoSavePrefs): Save any unsaved preferences before closing the window.

debug mode (debugMode): Enable features for debugging PsychoPy itself, including unit-tests.

locale (locale): Language to use in menus etc.; not all translations are available. Select a value, then restart the app. Think about [adding translations for your language](#).

2.4.2 Builder settings (Builder)

reload previous exp (reloadPrevExp): Select whether to automatically reload a previously opened experiment at start-up.

uncluttered namespace (unclutteredNamespace): If this option is selected, the scripts will use more complex code, but the advantage is that there is less of a chance that name conflicts will arise.

components folders (componentsFolders): A list of folder path names that can hold additional custom components for the Builder view; expects a comma-separated list.

hidden components (hiddenComponents): A list of components to hide (e.g., because you never use them)

unpacked demos dir (unpackedDemosDir): Location of Builder demos on this computer (after unpacking).

saved data folder (savedDataFolder): Name of the folder where subject data should be saved (relative to the script location).

Flow at top (topFlow): If selected, the “Flow” section will be shown topmost and the “Components” section will be on the left. Restart PsychoPy to activate this option.

always show readme (alwaysShowReadme): If selected, PsychoPy always shows the Readme file if you open an experiment. The Readme file needs to be located in the same folder as the experiment file.

max favorites (maxFavorites): Upper limit on how many components can be in the Favorites menu of the Components panel.

2.4.3 Coder settings (Coder)

code font (codeFont): A list of font names to be used for code display. The first found on the system will be used.

comment font (commentFont): A list of font names to be used for comments sections. The first found on the system will be used

output font (outputFont): A list of font names to be used in the output panel. The first found on the system will be used.

code font size (codeFontSize): An integer between 6 and 24 that specifies the font size for code display in points.

output font size (outputFontSize): An integer between 6 and 24 that specifies the font size for output display in points.

show source asst (showSourceAsst): Do you want to show the source assistant panel (to the right of the Coder view)? On Windows this provides help about the current function if it can be found. On OS X the source assistant is of limited use and is disabled by default.

show output (showOutput): Show the output panel in the Coder view. If shown all python output from the session will be output to this panel. Otherwise it will be directed to the original location (typically the terminal window that called PsychoPy application to open).

reload previous files (reloadPrevFiles): Should PsychoPy fetch the files that you previously had open when it launches?

preferred shell (preferredShell): Specify which shell should be used for the coder shell window.

newline convention (newlineConvention): Specify which character sequence should be used to encode newlines in code files: unix = n (line feed only), dos = rn (carriage return plus line feed).

2.4.4 General settings (General)

window type (winType): PsychoPy can use one of two ‘backends’ for creating windows and drawing; pygame and pyglet. Here you can set the default backend to be used.

units (units): Default units for windows and visual stimuli (‘deg’, ‘norm’, ‘cm’, ‘pix’). See *Units for the window and stimuli*. Can be overridden by individual experiments.

full-screen (fullscr): Should windows be created full screen by default? Can be overridden by individual experiments.

allow GUI (allowGUI): When the window is created, should the frame of the window and the mouse pointer be visible. If set to False then both will be hidden.

paths (paths): Paths for additional Python packages can be specified. See more information [here](#).

audio library (audioLib): As explained in the [Sound](#) documentation, currently two sound libraries are available, pygame and pyo.

audio driver (audioDriver): Also, different audio drivers are available.

flac audio compression (flac): Set flac audio compression.

parallel ports (parallelPorts): This list determines the addresses available in the drop-down menu for the [Parallel Port Out Component](#).

2.4.5 Connection settings (Connections)

proxy (proxy): The proxy server used to connect to the internet if needed. Must be of the form http://111.222.333.444:5555

auto-proxy (autoProxy): PsychoPy should try to deduce the proxy automatically. If this is True and autoProxy is successful, then the above field should contain a valid proxy address.

allow usage stats (allowUsageStats): Allow PsychoPy to ping a website at when the application starts up. Please leave this set to True. The info sent is simply a string that gives the date, PsychoPy version and platform info. There is no cost to you: no data is sent that could identify you and PsychoPy will not be delayed in starting as a result. The aim is simple: if we can show that lots of people are using PsychoPy there is a greater chance of it being improved faster in the future.

check for updates (checkForUpdates): PsychoPy can (hopefully) automatically fetch and install updates. This will only work for minor updates and is still in a very experimental state (as of v1.51.00).

timeout (timeout): Maximum time in seconds to wait for a connection response.

2.4.6 Key bindings

There are many shortcut keys that you can use in PsychoPy. For instance did you realise that you can indent or outdent a block of code with Ctrl-[and Ctrl-] ?

2.5 Data outputs

There are a number of different forms of output that PsychoPy can generate, depending on the study and your preferred analysis software. Multiple file types can be output from a single experiment (e.g. *Excel data file* for a quick browse, *Log file* to check for error messages and *PsychoPy data file (.psydat)* for detailed analysis)

2.5.1 Log file

Log files are actually rather difficult to use for data analysis but provide a chronological record of everything that happened during your study. The level of content in them depends on you. See *Logging data* for further information.

2.5.2 PsychoPy data file (.psydat)

This is actually a *TrialHandler* or *StairHandler* object that has been saved to disk with the python *cPickle* module.

These files are designed to be used by experienced users with previous experience of python and, probably, matplotlib. The contents of the file can be explored with *dir()*, as any other python object.

These files are ideal for batch analysis with a python script and plotting via *matplotlib*. They contain more information than the Excel or csv data files, and can even be used to (re)create those files.

Of particular interest might be the attributes of the Handler:

extraInfo the *extraInfo* dictionary provided to the Handler during its creation

trialList the list of dictionaries provided to the Handler during its creation

data a dictionary of 2D numpy arrays. Each entry in the dictionary represents a type of data (e.g. if you added 'rt' data during your experiment using `~psychopy.data.TrialHandler.addData` then 'rt' will be a key). For each of those entries the 2D array represents the condition number and repeat number (remember that these start at 0 in python, unlike Matlab(TM) which starts at 1)

For example, to open a psydat file and examine some of its contents with:

```
from psychopy.misc import fromFile
datFile = fromFile('fileName.psydat')
#get info (added when the handler was created)
print datFile.extraInfo
#get data
print datFile.data
#get list of conditions
conditions = datFile.trialList
for condN, condition in enumerate(conditions):
    print condition, datFile.data['response'][condN], numpy.mean(datFile.data['response'][condN])
```

Ideally, we should provide a demo script here for fetching and plotting some data (feel free to *contribute*).

2.5.3 Long-wide data file

This form of data file is the default data output from Builder experiments as of v1.74.00. Rather than summarising data in a spreadsheet where one row represents all the data from a single condition (as in the summarised data format), in long-wide data files the data is not collapsed by condition, but written chronologically with one row representing one trial (hence it is typically longer than summarised data files). One column in this format is used for every single piece of information available in the experiment, even where that information might be considered redundant (hence the format is also ‘wide’).

Although these data files might not be quite as easy to read quickly by the experimenter, they are ideal for import and analysis under packages such as R, SPSS or Matlab.

2.5.4 Excel data file

Excel 2007 files (.xlsx) are a useful and flexible way to output data as a spreadsheet. The file format is open and supported by nearly all spreadsheet applications (including older versions of Excel and also OpenOffice). N.B. because .xlsx files are widely supported, the older Excel file format (.xls) is not likely to be supported by PsychoPy unless a user contributes the code to the project.

Data from PsychoPy are output as a table, with a header row. Each row represents one condition (trial type) as given to the *TrialHandler*. Each column represents a different type of data as given in the header. For some data, where there are multiple columns for a single entry in the header. This indicates multiple trials. For example, with a standard data file in which response time has been collected as ‘rt’ there will be a heading *rt_raw* with several columns, one for each trial that occurred for the various trial types, and also an *rt_mean* heading with just a single column giving the mean reaction time for each condition.

If you’re creating experiments by writing scripts then you can specify the sheet name as well as file name for Excel file outputs. This way you can store multiple sessions for a single subject (use the subject as the filename and a date-stamp as the sheetname) or a single file for multiple subjects (give the experiment name as the filename and the participant as the sheetname).

Builder experiments use the participant name as the file name and then create a sheet in the Excel file for each loop of the experiment. e.g. you could have a set of practice trials in a loop, followed by a set of main trials, and these would each receive their own sheet in the data file.

2.5.5 Delimited text files (.csv, .tsv, .txt)

For maximum compatibility, especially for legacy analysis software, you can choose to output your data as a delimited text file. Typically this would be comma-separated values (.csv file) or tab-delimited (.tsv file). The format of those files is exactly the same as the Excel file, but is limited by the file format to a single sheet.

2.6 Gamma correcting a monitor

Monitors typically don't have linear outputs; when you request luminance level of 127, it is not exactly half the luminance of value 254. For experiments that require the luminance values to be linear, a correction needs to be put in place for this nonlinearity which typically involves fitting a power law or gamma (γ) function to the monitor output values. This process is often referred to as gamma correction.

PsychoPy can help you perform gamma correction on your monitor, especially if you have one of the supported photometers/spectroradiometers.

There are various different equations with which to perform gamma correction. The simple equation (2.1) is assumed by most hardware manufacturers and gives a reasonable first approximation to a linear correction. The full gamma correction equation (2.3) is more general, and likely more accurate especially where the lowest luminance value of the monitor is bright, but also requires more information. It can only be used in labs that do have access to a photometer or similar device.

2.6.1 Simple gamma correction

The simple form of correction (as used by most hardware and software) is this:

$$L(V) = a + kV^\gamma \quad (2.1)$$

where L is the final luminance value, V is the requested intensity (ranging 0 to 1), a , k and γ are constants for the monitor.

This equation assumes that the luminance where the monitor is set to 'black' ($V=0$) comes entirely from the surround and is therefore not subject to the same nonlinearity as the monitor. If the monitor itself contributes significantly to a then the function may not fit very well and the correction will be poor.

The advantage of this function is that the calibrating system (PsychoPy in this case) does not need to know anything more about the monitor than the gamma value itself (for each gun). For the full gamma equation (2.3), the system needs to know about several additional variables. The look-up table (LUT) values required to give a (roughly) linear luminance output can be generated by:

$$LUT(V) = V^{1/\gamma} \quad (2.2)$$

where V is the entry in the LUT, between 0 (black) and 1 (white).

2.6.2 Full gamma correction

For very accurate gamma correction PsychoPy uses a more general form of the equation above, which can separate the contribution of the monitor and the background to the lowest luminance level:

$$L(V) = a + (b + kV)^\gamma \quad (2.3)$$

This equation makes no assumption about the origin of the base luminance value, but requires that the system knows the values of b and k as well as γ .

The inverse values, required to build the LUT are found by:

$$LUT(V) = \frac{((1 - V)b^\gamma + V(b + k)^\gamma)^{1/\gamma} - b}{k} \quad (2.4)$$

This is derived below, for the interested reader. ;-)

And the associated luminance values for each point in the LUT are given by:

$$L(V) = a + (1 - V)b^\gamma + V(b + k)^\gamma$$

2.6.3 Deriving the inverse full equation

The difficulty with the full gamma equation (2.3) is that the presence of the b value complicates the issue of calculating the inverse values for the LUT. The simple inverse of (2.3) as a function of output luminance values is:

$$LUT(L) = \frac{((L - a)^{1/\gamma} - b)}{k} \quad (2.5)$$

To use this equation we need to first calculate the linear set of luminance values, L , that we are able to produce the current monitor and lighting conditions and *then* deduce the LUT value needed to generate that luminance value.

We need to insert into the LUT the values between 0 and 1 (to use the maximum range) that map onto the linear range from the minimum, m , to the maximum M possible luminance. From the parameters in (2.3) it is clear that:

$$\begin{aligned} m &= a + b^\gamma \\ M &= a + (b + k)^\gamma \end{aligned} \quad (2.6)$$

Thus, the luminance value, L at any given point in the LUT, V , is given by

$$\begin{aligned} L(V) &= m + (M - m)V \\ &= a + b^\gamma + (a + (b + k)^\gamma - a - b^\gamma)V \\ &= a + b^\gamma + ((b + k)^\gamma - b^\gamma)V \\ &= a + (1 - V)b^\gamma + V(b + k)^\gamma \end{aligned} \quad (2.7)$$

where V is the position in the LUT as a fraction.

Now, to generate the LUT as needed we simply take the inverse of (2.3):

$$LUT(L) = \frac{(L - a)^{1/\gamma} - b}{k} \quad (2.8)$$

and substitute our $L(V)$ values from (2.7):

$$\begin{aligned} LUT(V) &= \frac{(a + (1 - V)b^\gamma + V(b + k)^\gamma - a)^{1/\gamma} - b}{k} \\ &= \frac{((1 - V)b^\gamma + V(b + k)^\gamma)^{1/\gamma} - b}{k} \end{aligned} \quad (2.9)$$

2.6.4 References

2.7 OpenGL and Rendering

All rendering performed by PsychoPy uses hardware-accelerated OpenGL rendering where possible. This means that, as much as possible, the necessary processing to calculate pixel values is performed by the graphics card *GPU* rather than by the *CPU*. For example, when an image is rotated the calculations to determine what pixel values should result, and any interpolation that is needed, are determined by the graphics card automatically.

In the double-buffered system, stimuli are initially drawn into a piece of memory on the graphics card called the ‘back buffer’, while the screen presents the ‘front buffer’. The back buffer initially starts blank (all pixels are set to the window’s defined color) and as stimuli are ‘rendered’ they are gradually added to this back buffer. The way in which stimuli are combined according to transparency rules is determined by the *blend mode* of the window. At some point in time, when we have rendered to this buffer all the objects that we wish to be presented, the buffers are ‘flipped’ such that the stimuli we have been drawing are presented simultaneously. The monitor updates at a very precise fixed rate and the flipping of the window will be synchronised to this monitor update if possible (see *Sync to VBL and wait for VBL*).

Each update of the window is referred to as a ‘frame’ and this ultimately determines the temporal resolution with which stimuli can be presented (you cannot present your stimulus for any duration other than a multiple of the frame duration). In addition to synchronising flips to the frame refresh rate, PsychoPy can optionally go a further step of not allowing the code to continue until a screen flip has occurred on the screen, which is useful in ascertaining exactly when the frame refresh occurred (and, thus, when your stimulus actually appeared to the subject). These timestamps are very precise on most computers. For further information about synchronising and waiting for the refresh see [Sync to VBL and wait for VBL](#).

If the code/processing required to render all your stimuli to the screen takes longer to complete than one screen refresh then you will ‘drop/skip a frame’. In this case the previous frame will be left on screen for a further frame period and the flip will only take effect on the following screen update. As a result, time-consuming operations such as disk accesses or execution of many lines of code, should be avoided while stimuli are being dynamically updated (if you care about the precise timing of your stimuli). For further information see the sections on [Detecting dropped frames](#) and [Reducing dropped frames](#).

2.7.1 Fast and slow functions

The fact that modern graphics processors are extremely powerful; they can carry out a great deal of processing from a very small number of commands. Consider, for instance, the PsychoPy Coder demo *elementArrayStim* in which several hundred Gabor patches are updated frame by frame. The graphics card has to blend a sinusoidal grating with a grey background, using a Gaussian profile, several hundred times each at a different orientation and location and it does this in less than one screen refresh on a good graphics card.

There are three things that are relatively slow and should be avoided at critical points in time (e.g. when rendering a dynamic or brief stimulus). These are:

1. disk accesses
2. passing large amounts of data to the graphics card
3. making large numbers of python calls.

Functions that are very fast:

1. Calls that move, resize, rotate your stimuli are likely to carry almost no overhead
2. Calls that alter the color, contrast or opacity of your stimulus will also have no overhead IF your graphics card supports *OpenGL Shaders*
3. Updating of stimulus parameters for `psychopy.visual.ElementArrayStim` is also surprisingly fast BUT you should try to update your stimuli using *numpy* arrays for the maths rather than *for...* loops

Notable slow functions in PsychoPy calls:

1. Calls to set the image or set the mask of a stimulus. This involves having to transfer large amounts of data between the computer’s main processor and the graphics card, which is a relatively time-consuming process.
2. Any of your own code that uses a Python *for...* loop is likely to be slow if you have a large number of cycles through the loop. Try to ‘vectorise’ your code using a *numpy* array instead.

2.7.2 Tips to render stimuli faster

1. Keep images as small as possible. This is meant in terms of **number of pixels**, not in terms of Mb on your disk. Reducing the size of the image on your disk might have been achieved by image compression such as using jpeg images but these introduce artefacts and do nothing to reduce the problem of send large amounts of data from the CPU to the graphics card. Keep in mind the size that the image will appear on your monitor and how many pixels it will occupy there. If you took your photo using a 10 megapixel camera that means the image is

represented by 30 million numbers (a red, green and blue) but your computer monitor will have, at most, around 2 megapixels (1960x1080).

2. Try to use square powers of two for your image sizes. This is efficient because computer memory is organised according to powers of two (did you notice how often numbers like 128, 512, 1024 seem to come up when you buy your computer?). Also several mathematical routines (anything involving Fourier maths, which is used a lot in graphics processing) are faster with power-of-two sequences. For the `psychopy.visual.GratingStim` a texture/mask of this size is **required** and if you don't provide one then your texture will be 'upsampled' to the next larger square-power-of-2, so you can save this interpolation step by providing it in the right shape initially.
3. Get a faster graphics card. Upgrading to a more recent card will cost around £30. If you're currently using an integrated Intel graphics chip then almost any graphics card will be an advantage. Try to get an nVidia or an ATI Radeon card.

2.7.3 OpenGL Shaders

You may have heard mention of 'shaders' on the users mailing list and wondered what that meant (or maybe you didn't wonder at all and just went for a donut!). OpenGL shader programs allow modern graphics cards to make changes to things during the rendering process (i.e. while the image is being drawn). To use this you need a graphics card that supports OpenGL 2.1 and PsychoPy will only make use of shaders if a specific OpenGL extension that allows floating point textures is also supported. Nowadays nearly all graphics cards support these features - even Intel chips from Intel!

One example of how such shaders are used is the way that PsychoPy colors greyscale images. If you provide a greyscale image as a 128x128 pixel texture and set its color to be red then, without shaders, PsychoPy needs to create a texture that contains the 3x128x128 values where each of the 3 planes is scaled according to the RGB values you require. If you change the color of the stimulus a new texture has to be generated with the new weightings for the 3 planes. However, with a shader program, that final step of scaling the texture value according to the appropriate RGB value can be done by the graphics card. That means we can upload just the 128x128 texture (taking 1/3 as much time to upload to the graphics card) and then we each time we change the color of the stimulus we just a new RGB triplet (only 3 numbers) without having to recalculate the texture. As a result, on graphics cards that support shaders, changing colors, contrasts and opacities etc. has almost zero overhead.

2.7.4 Blend Mode

A 'blend function' determines how the values of new pixels being drawn should be combined with existing pixels in the 'frame buffer'.

blendMode = 'avg'

This mode is exactly akin to the real-world scenario of objects with varying degrees of transparency being placed in front of each other; increasingly transparent objects allow increasing amounts of the underlying stimuli to show through. Opaque stimuli will simply occlude previously drawn objects. With each increasing semi-transparent object to be added, the visibility of the first object becomes increasingly weak. The order in which stimuli are rendered is very important since it determines the ordering of the layers. Mathematically, each pixel colour is constructed from $\text{opacity} * \text{stimRGB} + (1 - \text{opacity}) * \text{backgroundRGB}$. This was the only mode available before PsychoPy version 1.80 and remains the default for the sake of backwards compatibility.

blendMode = 'add'

If the window *blendMode* is set to 'add' then the value of the new stimulus does not in any way *replace* that of the existing stimuli that have been drawn; it is added to it. In this case the value of *opacity* still affects the weighting of

the new stimulus being drawn but the first stimulus to be drawn is never ‘occluded’ as such. The sum is performed using the signed values of the color representation in PsychoPy, with the mean grey being represented by zero. So a dark patch added to a dark background will get even darker. For grating stimuli this means that contrast is summed correctly.

This blend mode is ideal if you want to test, for example, the way that subjects perceive the sum of two potentially overlapping stimuli. It is also needed for rendering stereo/dichoptic stimuli to be viewed through colored anaglyph glasses.

If stimuli are combined in such a way that an impossible luminance value is requested of any of the monitor guns then that pixel will be out of bounds. In this case the pixel can either be clipped to provide the nearest possible colour, or can be artificially colored with noise, highlighting the problem if the user would prefer to know that this has happened.

2.7.5 Sync to VBL and wait for VBL

PsychoPy will always, if the graphics card allows it, synchronise the flipping of the window with the vertical blank interval (VBL aka VBI) of the screen. This prevents visual artefacts such as ‘tearing’ of moving stimuli. This does not, itself, indicate that the script also waits for the physical frame flip to occur before continuing. If the *waitBlanking* window argument is set to False then, although the window refreshes themselves will only occur in sync with the screen VBL, the *win.flip()* call will not actually wait for this to occur, such that preparations can continue immediately for the next frame. For rendering purposes this is actually optimal and will reduce the likelihood of frames being dropped during rendering.

By default the PsychoPy Window will also wait for the VBL (*waitBlanking=True*) . Although this is slightly less efficient for rendering purposes it is necessary if we need to know exactly when a frame flip occurred (e.g. to timestamp when the stimulus was physically presented). On most systems this will provide a very accurate measure of when the stimulus was presented (with a variance typically well below 1ms but this should be tested on your system).

2.8 Projects

As of version 1.84 PsychoPy connects directly with the Open Science Framework website (<http://OSF.io>) allowing you to search for existing projects and upload your own experiments and data.

There are several reasons you may want to do this:

- sharing files with collaborators
- sharing files with the rest of the scientific community
- maintaining historical evidence of your work
- providing yourself with a simple version control across your different machines

2.8.1 Sharing with collaborators

You may find it simple to share files with your collaborators using dropbox but that means your data are stored by a commercial company over which you have no control and with no interest in scientific integrity. Check with your ethics committee how they feel about your data (e.g. personal details of participants?) being stored on dropbox. OSF, by comparison, is designed for scientists to store their data securely and forever.

Once you’ve created a project on OSF you can add other contributors to it and when they log in via PsychoPy they will see the projects they share with you (as well as the project they have created themselves). Then they can sync with that project just like any other.

2.8.2 Sharing files/projects with others

Optionally, you can make your project (or subsets of it) publicly accessible so that others can view the files. This has various advantages, to the scientific field but also to you as a scientist.

Good for open science:

- Sharing your work allows scientists to work out why one experiment gave a different result to another; there are often subtleties in the exact construction of a study that didn't get described fully in the methods section. By sharing the actual experiment, rather than just a description of it, we can reduce the failings of replications
- Sharing your work helps others get up and running quickly. That's good for the scientific community. We want science to progress faster and with fewer mistakes.

Some people feel that, having put in all that work to create their study, it would be giving up their advantage to let others simply use their work. Luckily, sharing is good for you as a scientist as well!

Good for the scientist:

- When you create a study you want others to base their work on yours (we call that academic impact)
- By giving people the exact materials from your work you increase the chance that they will work on your topic and base their next study on something of yours
- By making your project publicly available on OSF (or other sharing repository) you raise visibility of your work

You don't need to decide to share immediately. Probably you want your work to be private until the experiment is complete and the paper is under review (or has been accepted even). That's fine. You can create your project and keep it private between you and your collaborators and then share it at a later date with the click of a button.

2.8.3 Maintaining a validated history of your work

In many areas of science researchers are very careful about maintaining a full documented history of what their work; what the

- you can “preregister” your plans for the next experiment (so that people can't later accuse you of “p-hacking”).
- all your files are timestamped so you can prove to others that they were collected on/by a certain date, removing any potential doubts about who collected data first
- your projects (and individual files) have a unique URL on OSF so you can cite/reference resources.

Additionally, “Registrations” (snapshots of your project at a fixed point in time) can be given a DOI, which guarantees they will exist permanently

2.8.4 Using PsychoPy to sync with OSF

PsychoPy doesn't currently have the facility to *create* user profiles or projects, so the first step is for you to do that yourself.

Login to OSF

From the *Projects* menu you can log in to OSF with your username and password (this is never stored; see [Security](#)). This user will stay logged in while the PsychoPy application remains open, or until you switch to a different user. If

you select “Remember me” then your login will be stored and you can log in again without typing your password each time.

Projects that you have previously synchronised will try to use the stored details of the known users if possible and will revert to username and password if not. Project files (defining the details of the project to sync) can be stored wherever you choose; either in a private or shared location. User details are stored in the home space of the user currently logged in to the operating system so are not shared with other users by default.

Security

When you log in with your username and password these details are not stored by PsychoPy in any way. They are sent immediately to OSF using a secure (https) connection. OSF sends back an “authorisation token” identifying you as a valid user with authorised credentials. This is stored locally for future log in attempts. By visiting your user profile at <http://OSF.io> you can see what applications/computers have retrieved authorisation tokens for your account (and revoke them if you choose).

The auth token is stored in plain text on your computer, but a malicious attacker with access to your computer could only use this to log in to OSF.io. They could not use it to work out your password.

All files are sent by secure connection (https) to the server.

Searching for projects

Having logged in to OSF from the projects menu you can search for projects to work with using the *>Projects>Search* menu. This brings up a view that shows you all the current projects for the logged in user (owned or shared) and allows you to search for public projects using tags and/or words in the title.

When you select a project, either in your own projects or in the search box, then the details for that project come up on the right hand side, including a link to visit the project page on the web site.

On the web page for the project you can “fork” the project to your own username and then you can use PsychoPy to download/update/sync files with that project, just as with any other project. The project retains information about its history; the project from which it was forked gets its due credit.

Synchronizing projects

Having found your project online you can then synchronize a local folder with that set of files.

To do this the first time:

- select one of your projects in the project search window so the details appear on the right
- press the “Sync...” button
- the Project Sync dialog box will appear
- set the location/name for a project file, which will store information about the state of files on the last sync
- set the location of the (root) folder locally that you want to be synchronised with the remote files
- press sync

The sync process and rules:

- on the first synchronisation all the files/folders will be merged: - the contents of the local folder will be uploaded to the server and vice versa - files that have the same name but different contents (irrespective of dates) will be flagged as conflicting (see below) and both copies kept

- on subsequent sync operations a two-way sync will be performed taking into account the previous state. **If you delete the files locally and then sync then they will be deleted remotely as well**
- files that are the same (according to an md5 checksum) and have the same location will be left as they are
- if a file is in conflict (it has been changed in both locations since the last sync) then both versions will be kept and will be tagged as conflicting
- if a file is deleted in one location but is also changed in the other (since the last sync) then it will be recreated on the side where it was deleted with the state of the side where it was not deleted.

Conflicting files will be labelled with their original filename plus the string “_CONFLICT<timestamp>” Deletion conflicts will be labelled with their original filename plus the string “_DELETED”

Limitations

- PsychoPy does not directly allow you to create a new project yet, nor create a user account. To start with you need to go to <http://osf.io> to create your username and/or project. You also cannot currently fork public projects to your own user space yet from within PsychoPy. If you find a project that is useful to you then fork it from the website (the link is available in the details panel of the project search window)
- The synchronisation routines are fairly basic right now and will not cater for all possible eventualities. For example, if you create a file locally but your colleague created a folder with the same name and synced that with the server, it isn't clear what will (or should ideally) happen when you now sync your project. You should be careful with this tool and always back up your data by an independent means in case damage to your files is caused
- This functionality is new and may well have bugs. **User beware!**

2.9 Timing Issues and synchronisation

One of the key requirements of experimental control software is that it has good temporal precision. PsychoPy aims to be as precise as possible in this domain and can achieve excellent results depending on your experiment and hardware. It also provides you with a precise log file of your experiment to allow you to check the precision with which things occurred. Some general considerations are discussed here and there are links with *Specific considerations for specific designs*.

Something that people seem to forget (not helped by the software manufacturers that keep talking about their sub-millisecond precision) is that the monitor, keyboard and human participant DO NOT have anything like this sort of precision. Your monitor updates every 10-20ms depending on frame rate. If you use a CRT screen then the top is drawn before the bottom of the screen by several ms. If you use an LCD screen the whole screen can take around 20ms to switch from one image to the next. Your keyboard has a latency of 4-30ms, depending on brand and system.

So, yes, PsychoPy's temporal precision is as good as most other equivalent applications, for instance the duration for which stimuli are presented can be synchronised precisely to the frame, but the overall accuracy is likely to be severely limited by your experimental hardware. To get **very** precise timing of responses etc., you need to use specialised hardware like button boxes and you need to think carefully about the physics of your monitor.

Warning: The information about timing in PsychoPy assumes that your graphics card is capable of synchronising with the monitor frame rate. For integrated Intel graphics chips (e.g. GMA 945) under Windows, this is not true and the use of those chips is not recommended for serious experimental use as a result. Desktop systems can have a moderate graphics card added for around £30 which will be vastly superior in performance.

2.9.1 Specific considerations for specific designs

Non-slip timing for imaging

For most behavioural/psychophysics studies timing is most simply controlled by setting some timer (e.g. a `clock()`) to zero and waiting until it has reached a certain value before ending the trial. We might call this a ‘relative’ timing method, because everything is timed from the start of the trial/epoch. In reality this will cause an overshoot of some fraction of one screen refresh period (10ms, say). For imaging (EEG/MEG/fMRI) studies adding 10ms to each trial repeatedly for 10 minutes will become a problem, however. After 100 stimulus presentations your stimulus and scanner will be de-synchronised by 1 second.

There are two ways to get around this:

1. *Time by frames* If you are confident that you *aren't dropping frames* then you could base your timing on frames instead to avoid the problem.
2. *Non-slip (global) clock timing* The other way, which for imaging is probably the most sensible, is to arrange timing based on a global clock rather than on a relative timing method. At the start of each trial you add the (known) duration that the trial will last to a *global* timer and then wait until that timer reaches the necessary value. To facilitate this, the PsychoPy `clock()` was given a new `add()` method as of version 1.74.00 and a `CountdownTimer()` was also added.

The non-slip method can only be used in cases where the trial is of a known duration at its start. It cannot, for example, be used if the trial ends when the subject makes a response, as would occur in most behavioural studies.

Non-slip timing from the Builder

(new feature as of version 1.74.00)

When creating experiments in the *Builder*, PsychoPy will attempt to identify whether a particular *Routine* has a known endpoint in seconds. If so then it will use non-slip timing for this Routine based on a global countdown timer called *routineTimer*. Routines that are able to use this non-slip method are shown in green in the *Flow*, whereas Routines using relative timing are shown in red. So, if you are using PsychoPy for imaging studies then make sure that all the Routines within your loop of epochs are showing as green. (Typically your study will also have a Routine at the start waiting for the first scanner pulse and this will use relative timing, which is appropriate).

Detecting dropped frames

Occasionally you will drop frames if you:

- try to do too much drawing
- do it in an inefficient manner (write poor code)
- have a poor computer/graphics card

Things to avoid:

- recreating textures for stimuli
- building new stimuli from scratch (create them once at the top of your script and then change them using `stim.setOri(ori)()`, `stim.setPos([x,y]...)`)

Turn on frame time recording

The key sometimes is *knowing* if you are dropping frames. PsychoPy can help with that by keeping track of frame durations. By default, frame time tracking is turned off because many people don't need it, but it can be turned on any time after Window creation `setRecordFrameIntervals()`, e.g.:

```
from psychopy import visual win = visual.Window([800,600]) win.setRecordFrameIntervals(True)
```

Since there are often dropped frames just after the system is initialised, it makes sense to start off with a fixation period, or a ready message and don't start recording frame times until that has ended. Obviously if you aren't refreshing the window at some point (e.g. waiting for a key press with an unchanging screen) then you should turn off the recording of frame times or it will give spurious results.

Warn me if I drop a frame

The simplest way to check if a frame has been dropped is to get PsychoPy to report a warning if it thinks a frame was dropped:

```
from psychopy import visual, logging
win = visual.Window([800,600])
win.setRecordFrameIntervals(True)
win._refreshThreshold=1/85.0+0.004 #i've got 85Hz monitor and want to allow 4ms tolerance
#set the log module to report warnings to the std output window (default is errors only)
logging.console.setLevel(logging.WARNING)
```

Show me all the frame times that I recorded

While recording frame times, these are simply appended, every frame to `win.frameIntervals` (a list). You can simply plot these at the end of your script using `pylab`:

```
import pylab
pylab.plot(win.frameIntervals)
pylab.show()
```

Or you could save them to disk. A convenience function is provided for this:

```
win.saveFrameIntervals(fileName=None, clear=True)
```

The above will save the currently stored frame intervals (using the default filename, 'lastFrameIntervals.log') and then clears the data. The saved file is a simple text file.

At any time you can also retrieve the time of the /last/ frame flip using `win.lastFrameT` (the time is synchronised with `logging.defaultClock` so it will match any logging commands that your script uses).

'Blocking' on the VBI

As of version 1.62 PsychoPy 'blocks' on the vertical blank interval meaning that, once `Window.flip()` has been called, no code will be executed until that flip actually takes place. The timestamp for the above frame interval measurements is taken immediately after the flip occurs. Run the `timeByFrames` demo in `Coder` to see the precision of these measurements on your system. They should be within 1ms of your mean frame interval.

Note that Intel integrated graphics chips (e.g. GMA 945) under `win32` do not sync to the screen at all and so blocking on those machines is not possible.

Reducing dropped frames

There are many things that can affect the speed at which drawing is achieved on your computer. These include, but are probably not limited to; your graphics card, CPU, operating system, running programs, stimuli, and your code itself. Of these, the CPU and the OS appear to make rather little difference. To determine whether you are actually dropping frames see [Detecting dropped frames](#).

Things to change on your system:

1. make sure you have a good graphics card. Avoid integrated graphics chips, especially Intel integrated chips and especially on laptops (because on these you don't get to change your mind so easily later). In particular, try to make sure that your card supports OpenGL 2.0
2. **shut down as many programs, including background processes. Although modern processors are fast and often have multiple cores, they are still limited by the amount of RAM and the speed of the disk.**
 - anti-virus auto-updating (if you're allowed)
 - email checking software
 - file indexing software
 - backup solutions (e.g. TimeMachine)
 - Dropbox
 - Synchronisation software

Writing optimal scripts

1. run in full-screen mode (rather than simply filling the screen with your window). This way the OS doesn't have to spend time working out what application is currently getting keyboard/mouse events.
2. don't generate your stimuli when you need them. Generate them in advance and then just modify them later with the methods like `setContrast()`, `setOrientation()` etc...
3. **calls to the following functions are comparatively slow; they require more CPU time than most other functions and then have a large overhead.**
 - (a) `GratingStim.setTexture()`
 - (b) `RadialStim.setTexture()`
 - (c) `TextStim.setText()`
4. if you don't have OpenGL 2.0 then calls to `setContrast`, `setRGB` and `setOpacity` will also be slow, because they also make a call to `setTexture()`. If you have shader support then this call is not necessary and a large speed increase will result.
5. avoid loops in your python code (use numpy arrays to do maths with lots of elements)
6. if you need to create a large number (e.g. greater than 10) similar stimuli, then try the `ElementArrayStim`

Possible good ideas

It isn't clear that these actually make a difference, but they might).

1. disconnect the internet cable (to prevent programs performing auto-updates?)
2. on Macs you can actually shut down the Finder. It might help. See Alex Holcombe's page [here](#)

3. use a single screen rather than two (probably there is some graphics card overhead in managing double the number of pixels?)

Comparing Operating Systems under PsychoPy

This is an attempt to quantify the ability of PsychoPy draw without dropping frames on a variety of hardware/software. The following tests were conducted using the script at the bottom of the page. Note, of course that the hardware fully differs between the Mac and Linux/Windows systems below, but that both are standard off-the-shelf machines.

All of the below tests were conducted with ‘normal’ systems rather than anything that had been specifically optimised:

- the machines were connected to network
- did not have anti-virus turned off (except Ubuntu had no anti-virus)
- they even all had dropbox clients running
- Linux was the standard (not ‘realtime’ kernel)

No applications were actively being used by the operator while tests were run.

In order to test drawing under a variety of processing loads the test stimulus was one of:

- a single drifting Gabor
- 500 random dots continuously updating
- 750 random dots continuously updating
- 1000 random dots continuously updating

Common settings:

- Monitor was a CRT 1024x768 100Hz
- all tests were run in full screen mode with mouse hidden

System Differences:

- the iMac was lower spec than the Windows/Linux box and running across two monitors (necessary in order to connect to the CRT)
- the Windows/Linux box ran off a single monitor

Each run below gives the number of dropped frames out of a run of 10,000 (2.7 mins at 100Hz).

—	Windows XP	Windows 7	Mac OS X 10.6	Ubuntu 11.10
—	(SP3)	Enterprise	Snow Leopard	
Gabor	0	5	0	0
500-dot RDK	0	5	54	3
750-dot RDK	21	7	aborted	1174
1000-dot RDK	776	aborted	aborted	aborted
GPU	Radeon 5400	Radeon 5400	Radeon 2400	Radeon 5400
GPU driver	Catalyst 11.11	Catalyst 11.11		Catalyst 11.11
CPU	Core Duo 3GHz	Core Duo 3GHz	Core Duo 2.4GHz	Core Duo 3GHz
RAM	4GB	4GB	2GB	4GB

I’ll gradually try to update these tests to include:

- longer runs (one per night!)
- a faster Mac

- a real-time Linux kernel

2.9.2 Other questions about timing

Can PsychoPy deliver millisecond precision?

The simple answer is ‘yes’, given some additional hardware. The clocks that PsychoPy uses do have sub-millisecond precision but your keyboard has a latency of 4-25ms depending on your platform and keyboard. You could buy a response pad (e.g. a [Cedrus Response Pad](#)) and use PsychoPy’s serial port commands to retrieve information about responses and timing with a precision of around 1ms.

Before conducting your experiment in which effects might be on the order of 1 ms, do consider that;

- your screen has a temporal resolution of ~10 ms
- your visual system has a similar upper limit (or you would notice the flickering screen)
- human response times are typically in the range 200-400 ms and very variable
- USB keyboard latencies are variable, in the range 20-30ms

That said, PsychoPy does aim to give you as high a temporal precision as possible, and is likely not to be the limiting factor of your experiment.

Computer monitors

Monitors have fixed refresh rates, typically 60 Hz for a flat-panel display, higher for a CRT (85-100 Hz are common, up to 200 Hz is possible). For a refresh rate of 85 Hz there is a gap of 11.7 ms between frames and this limits the timing of stimulus presentation. You cannot have your stimulus appear for 100 ms, for instance; on an 85Hz monitor it can appear for either 94 ms (8 frames) or 105 ms (9 frames). There are further, less obvious, limitations however.

For ‘CRT (cathode ray tube) screens’, the lines of pixels are drawn sequentially from the top to the bottom and once the bottom line has been drawn the screen is finished and the line returns to the top (the Vertical Blank Interval, VBI). Most of your frame interval is spent drawing the lines with 1-2ms being left for the VBI. This means that the pixels at the bottom are drawn ‘up to 10 ms later’ than the pixels at the top of the screen. At what point are you going to say your stimulus ‘appeared’ to the participant? For flat panel displays, or (or LCD projectors) your image will be presented simultaneously all over the screen, but it takes up to 20 ms(!) for your pixels to go all the way from black to white (manufacturers of these panels quote values of 3 ms for the fastest panels, but they certainly don’t mean 3 ms white-to-black, I assume they mean 3 ms half-life).

Warning: If you’re using a regular computer display, *you have a hardware-limited temporal precision of 10 ms irrespective of your response box or software clocks etc...* and should bear that in mind when looking for effect sizes of less than that.

Can I have my stimulus to appear with a very precise rate?

Yes. Generally to do that you should time your stimulus (its onset/offset, its rate of change...) using the frame refresh rather than a clock. e.g. you should write your code to say ‘for 20 frames present this stimulus’ rather than ‘for 300ms present this stimulus’. Provided your graphics card is set to synchronise page-flips with the vertical blank, and provided that you aren’t [dropping frames](#) the frame rate will always be absolutely constant.

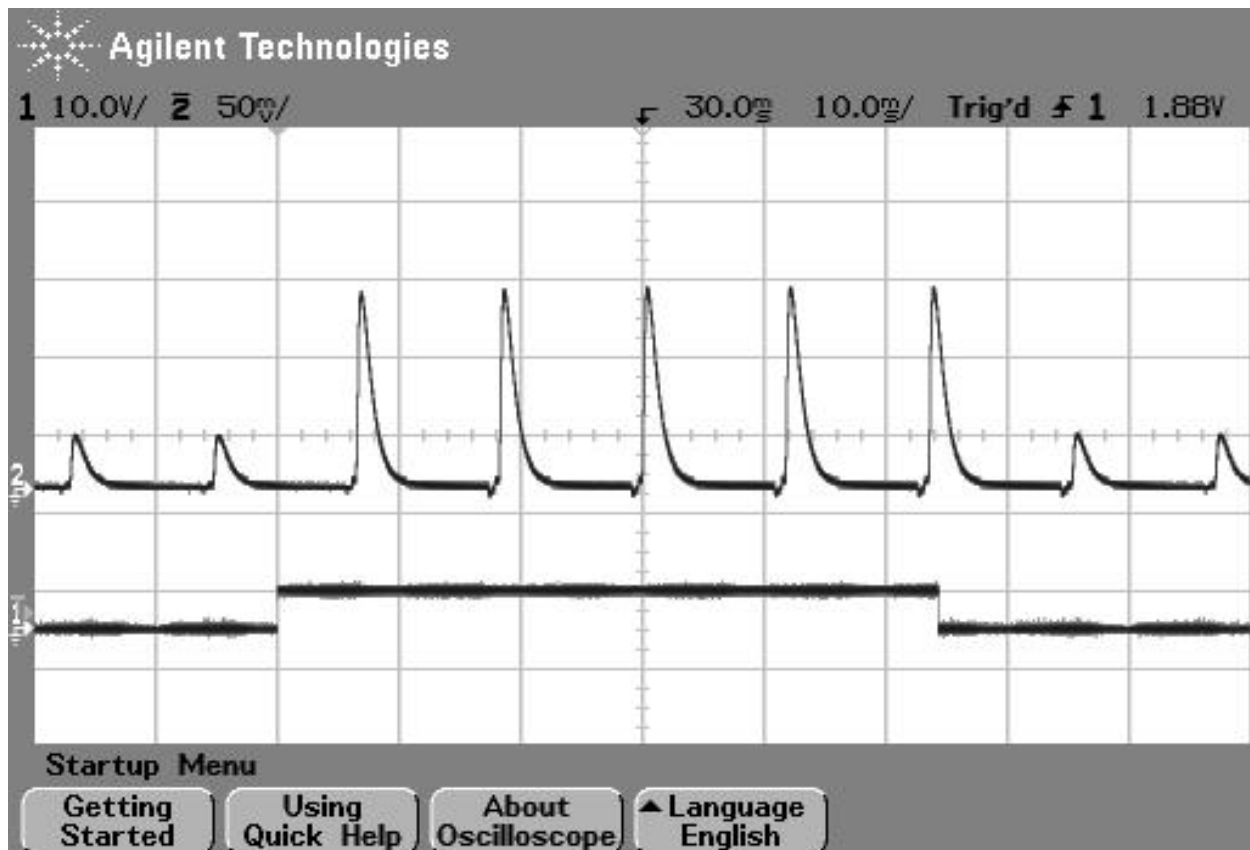


Fig. 2.1: Figure 1: photodiode trace at top of screen. The image above shows the luminance trace of a CRT recorded by a fast photo-sensitive diode at the top of the screen when a stimulus is requested (shown by the square wave). The square wave at the bottom is from a parallel port that indicates when the stimulus was flipped to the screen. Note that on a CRT the screen at any point is actually black for the majority of the time and just briefly bright. The visual system integrates over a large enough time window not to notice this. On the next frame after the stimulus ‘presentation time’ the luminance of the screen flash increased.

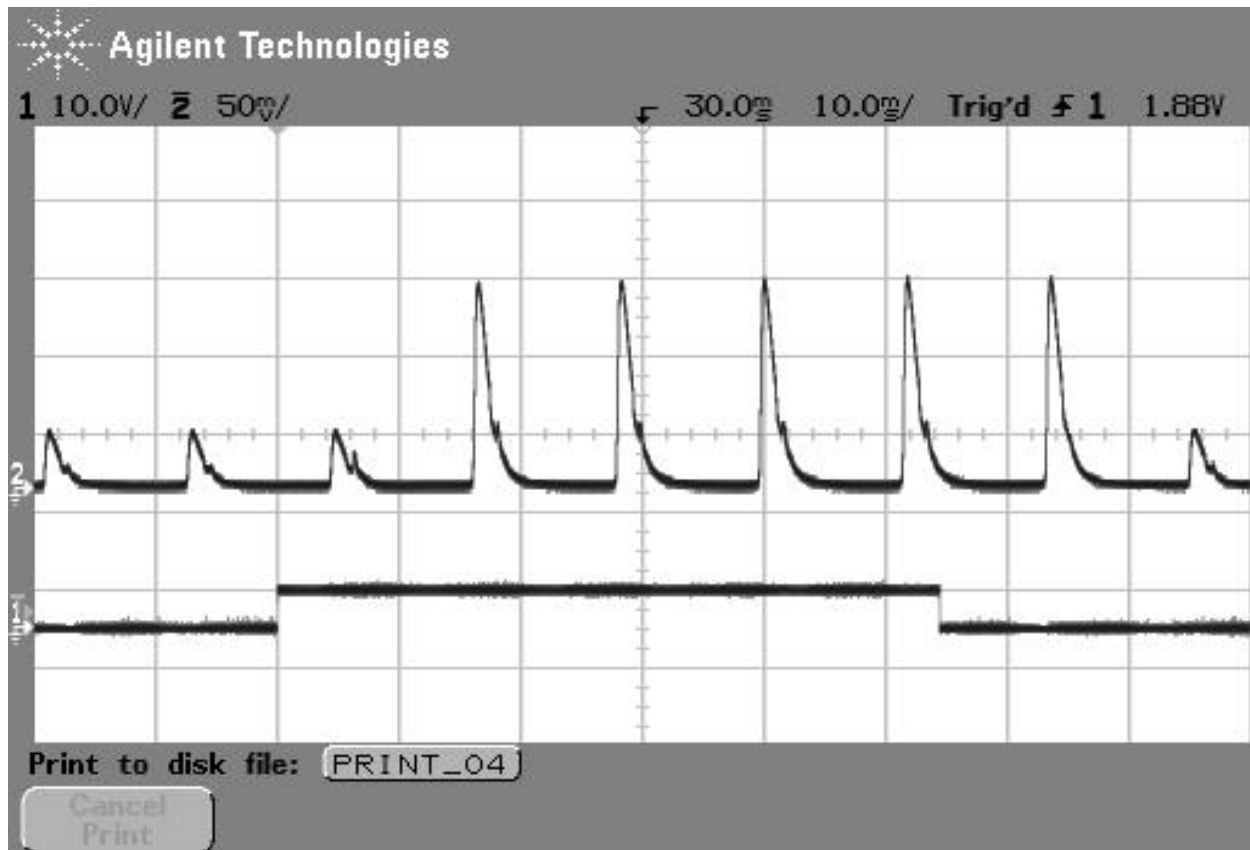


Fig. 2.2: Figure 2: photodiode trace of the same large stimulus at bottom of screen. The image above shows comes from exactly the same script as the above but the photodiode is positioned at the bottom of the screen. In this case, after the stimulus is 'requested' the current frame (which is dark) finishes drawing and then, 10ms later than the above image, the screen goes bright at the bottom.

2.10 Glossary

- Adaptive staircase** An experimental method whereby the choice of stimulus parameters is not pre-determined but based on previous responses. For example, the difficulty of a task might be varied trial-to-trial based on the participant's responses. These are often used to find psychophysical thresholds. Contrast this with the *method of constants*.
- CPU Central Processing Unit** is the main processor of your computer. This has a lot to do, so we try to minimise the amount of processing that is needed, especially during a trial, when time is tight to get the stimulus presented on every screen refresh.
- CRT Cathode Ray Tube** 'Traditional' computer monitor (rather than an LCD or plasma flat screen).
- csv Comma-Separated Value files** Type of basic text file with 'comma-separated values'. This type of file can be opened with most spreadsheet packages (e.g. MS Excel) for easy reading and manipulation.
- GPU Graphics Processing Unit** is the processor on your graphics card. The GPUs of modern computers are incredibly powerful and it is by allowing the GPU to do a lot of the work of rendering that PsychoPy is able to achieve good timing precision despite being written in an interpreted language
- Method of constants** An experimental method whereby the parameters controlling trials are predetermined at the beginning of the experiment, rather than determined on each trial. For example, a stimulus may be presented for 3 pre-determined time periods (100, 200, 300ms) on different trials, and then repeated a number of times. The order of presentation of the different conditions can be randomised or sequential (in a fixed order). Contrast this method with the *adaptive staircase*.
- VBI (Vertical Blank Interval, aka the Vertical Retrace, or Vertical Blank, VBL).** The period in-between video frames and can be used for synchronising purposes. On a CRT display the screen is black during the VBI and the display beam is returned to the top of the display.
- VBI blocking** The setting whereby all functions are synced to the VBI. After a call to `psychopy.visual.Window.flip()` nothing else occurs until the VBI has occurred. This is optimal and allows very precise timing, because as soon as the flip has occurred a very precise time interval is known to have occurred.
- VBI syncing** (aka vsync) The setting whereby the video drawing commands are synced to the VBI. When `psychopy.visual.Window.flip()` is called, the current back buffer (where drawing commands are being executed) will be held and drawn on the next VBI. This does not necessarily entail *VBI blocking* (because the system may return and continue executing commands) but does guarantee a fixed interval between frames being drawn.
- xlsx Excel OpenXML file format.** A spreadsheet data format developed by Microsoft but with an open (published) format. This is the native file format for Excel (2007 or later) and can be opened by most modern spreadsheet applications including OpenOffice (3.0+), google docs, Apple iWork 08.

Installation

3.1 Overview

PsychoPy can be installed in three main ways:

- **As an application:** The “Stand Alone” versions include everything you need to create and run experiments. When in doubt, choose this option.
- **As libraries:** PsychoPy and the libraries it depends on can also be installed individually, providing greater flexibility. This option requires managing a python environment.
- **As source code:** If you want to customize how PsychoPy works, consult the *developer’s guide* for installation and work-flow suggestions.

When you start PsychoPy for the first time, a **Configuration Wizard** will retrieve and summarize key system settings. Based on the summary, you may want to adjust some preferences to better reflect your environment. In addition, this is a good time to unpack the Builder demos to a location of your choice. (See the Demo menu in the Builder.)

If you get stuck or have questions, please [email the mailing list](#).

If all goes well, at this point your installation will be complete! See the next section of the manual, Getting started.

3.2 Recommended hardware

The minimum requirement for PsychoPy is a computer with a graphics card that supports OpenGL. Many newer graphics cards will work well. Ideally the graphics card should support OpenGL version 2.0 or higher. Certain visual functions run much faster if OpenGL 2.0 is available, and some require it (e.g. ElementArrayStim).

If you already have a computer, you can install PsychoPy and the Configuration Wizard will auto-detect the card and drivers, and provide more information. It is inexpensive to upgrade most desktop computers to an adequate graphics card. High-end graphics cards can be very expensive but are only needed for vision research (and high-end gaming).

If you’re thinking of buying a laptop for running experiments, **avoid the built-in Intel graphics chips (e.g. GMA 950)**. The drivers are crummy and performance is poor; graphics cards on laptops are more difficult to exchange. Get something with nVidia or ATI chips instead. Some graphics cards that are known to work with PsychoPy [can be found here](#); that list is not exhaustive, many cards will also work.

3.3 Windows

Once installed, you'll now find a link to the PsychoPy application in > Start > Programs > PsychoPy2. Click that and the Configuration Wizard should start.

The wizard will try to make sure you have reasonably current drivers for your graphics card. You may be directed to download the latest drivers from the vendor, rather than using the pre-installed Windows drivers. If necessary, get new drivers directly from the graphics card vendor; don't rely on Windows updates. The Windows-supplied drivers are buggy and sometimes don't support OpenGL at all.

The StandAlone installer adds the PsychoPy folder to your path, so you can run the included version of python from the command line. If you have your own version of python installed as well then you need to check which one is run by default, and change your path according to your personal preferences.

3.4 Mac OS X

There are different ways to install PsychoPy on a Mac that will suit different users. Almost all Mac's come with a suitable video card by default.

- Intel Mac users (with OS X v10.7 or higher; 10.5 and 10.6 might still work) can simply [download](#) the standalone application bundle (the **dmg** file) and drag it to their Applications folder. (Installing it elsewhere should work fine too.)
- Users of [macports](#) can install PsychoPy and all its dependencies simply with:

```
sudo port install py25-psychopy
```

(Thanks to James Kyles.)

- For PPC Macs (or for Intel Mac users that want their own custom python for running PsychoPy) you need to install the dependencies and PsychoPy manually. The easiest way is to use the *Enthought Python Distribution* (see Dependencies, below).
- You could alternatively manually install the 'framework build' of python and the dependencies (see below). One advantage to this is that you can then upgrade versions with:

```
sudo easy_install -N -Z -U psychopy
```

3.5 Linux

Debian systems: PsychoPy is in the Debian packages index so you can simply do:

```
sudo apt-get install psychopy
```

Ubuntu (and other Debian-based distributions):

1. Add the following sources in Synaptic, in the Configuration > Repository dialog box, under "Other software":

```
deb http://neuro.debian.net/debian karmic main contrib non-free
deb-src http://neuro.debian.net/debian karmic main contrib non-free
```

2. Then follow the 'Package authentication' procedure described in <http://neuro.debian.net/>
3. Then install the psychopy package under Synaptic or through `sudo apt-get install psychopy` which will install all dependencies.

(Thanks to Yaroslav Halchenko for the Debian and NeuroDebian package.)

Gentoo PsychoPy is in the Gentoo Sceince Overlay (see [this link](#) for the ebuild files). After you have [enabled the overlay](#) simply run:

```
# emerge psychopy
```

Other systems: You need to install the dependencies below. Then install PsychoPy:

```
$ sudo easy_install psychopy
...
Downloading http://psychopy.googlecode.com/files/PsychoPy-1.75.01-py2.7.egg
```

Dependencies

Like many open-source programs, PsychoPy depends on the work of many other people in the form of libraries.

4.1 Essential packages

Python: If you need to install python, or just want to, the easiest way is to use the [Enthought Python Distribution](#), which is [free for academic use](#). Be sure to get a 32-bit version. The only things it misses are *avbin*, *pyo*, and *flac*.

If you want to install each library individually rather than use the simpler distributions of packages above then you can download the following. Make sure you get the correct version for your OS and your version of Python. `easy_install` will work for many of these, but some require compiling from source.

- [python](#) (32-bit only, version 2.6 or 2.7; 2.5 might work, 3.x will not)
- [avbin](#) (movies) On Mac: 1) Download version 5 [from Google](#) (not a higher version). 2) Start terminal, type `sudo mkdir -p /usr/local/lib`. 3) `cd` to the unpacked avbin directory, type `sh install.sh`. 4) Start or restart PsychoPy, and from PsychoPy's coder view shell, this should work: `from pyglet.media import avbin`. If you run a script and get an error saying `'NoneType' object has no attribute 'blit'`, it probably means you did not install version 5.
- [setuptools](#)
- [numpy](#) (version 0.9.6 or greater)
- [scipy](#) (version 0.4.8 or greater)
- [pyglet](#) (version 1.1.4, not version 1.2)
- [wxPython](#) (version 2.8.10 or 2.8.11, not 2.9)
- [Python Imaging Library](#) (`sudo easy_install PIL`)
- [matplotlib](#) (for plotting and fast polygon routines)
- [lxml](#) (needed for loading/saving builder experiment files)
- [openpyxl](#) (for loading params from xlsx files)
- [pyo](#) (sound, version 0.6.2 or higher, compile with `—no-messages`)

These packages are only needed for Windows:

- [pywin32](#)
- [winioport](#) (to use the parallel port)
- [inpout32](#) (an alternative method to using the parallel port on Windows)

- `inpoutx64` (to use the parallel port on 64-bit Windows)

These packages are only needed for Linux:

- `pyparallel` (to use the parallel port)

4.2 Suggested packages

In addition to the required packages above, additional packages can be useful to PsychoPy users, e.g. for controlling hardware and performing specific tasks. These are packaged with the Standalone versions of PsychoPy but users with their own custom Python environment need to install these manually. Most of these can be installed with *easy_install*.

General packages:

- `psignifit` for bootstrapping and other resampling tests
- `pyserial` for interfacing with the serial port
- `parallel python` (aka `pp`) for parallel processing
- `flac` audio codec, for working with google-speech

Specific hardware interfaces:

- `pynetstation` to communicate with EGI netstation. See notes on using *egi* (*pynetstation*)
- `ioLabs` toolbox
- `labjack` toolbox

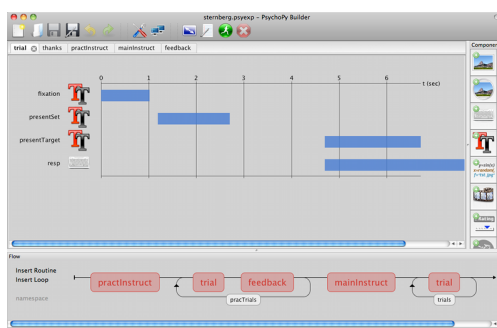
For developers:

- `pytest` and `coverage` for running unit tests
- `sphinx` for building documentation

Getting Started

As an application, PsychoPy has two main views: the [Builder](#) view, and the [Coder](#) view. It also has a underlying [API](#) that you can call directly.

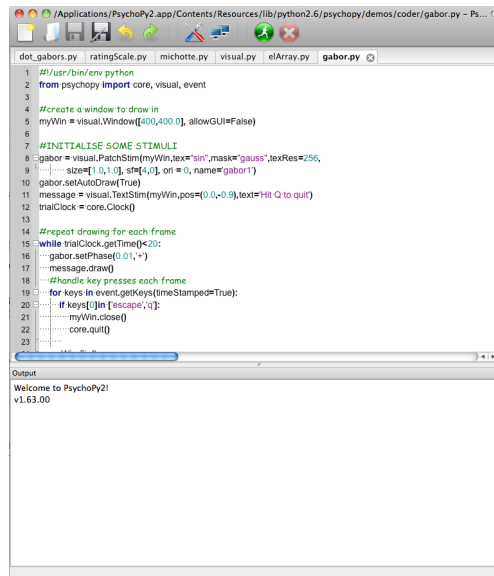
1. **Builder.** You can generate a wide range of experiments easily from the Builder using its intuitive, graphical user interface (GUI). This might be all you ever need to do. But you can always compile your experiment into a python script for fine-tuning, and this is a quick way for experienced programmers to explore some of PsychoPy's libraries and conventions.



2. **Coder.** For those comfortable with programming, the Coder view provides a basic code editor with syntax highlighting, code folding, and so on. Importantly, it has its own output window and Demo menu. The demos illustrate how to do specific tasks or use specific features; they are not whole experiments. The [Coder tutorials](#) should help get you going, and the [API reference](#) will give you the details.

The Builder and Coder views are the two main aspects of the PsychoPy application. If you've installed the StandAlone version of PsychoPy on **MS Windows** then there should be an obvious link to PsychoPy in your > Start > Programs. If you installed the StandAlone version on **Mac OS X** then the application is where you put it (!). On these two platforms you can open the Builder and Coder views from the View menu and the default view can be set from the preferences. **On Linux**, you can start PsychoPy from a command line, or make a launch icon (which can depend on the desktop and distro). If the PsychoPy app is started with flags `—coder` (or `-c`), or `—builder` (or `-b`), then the preferences will be overridden and that view will be created as the app opens.

For experienced python programmers, it's possible to use PsychoPy without ever opening the Builder or Coder. Install the PsychoPy libraries and dependencies, and use your favorite IDE instead of the Coder.



```
1 #!/usr/bin/env python
2 from psychopy import core, visual, event
3
4 #create a window to draw in
5 myWin = visual.Window([400,400], allowGUI=False)
6
7 #INITIALISE SOME STIMULI
8 gabor = visual.PatchSim(myWin, tex='sin', mask='gauss', texRes=256,
9 size=[1,1], sf=[4,0], ori=0, name='gabor1')
10 gabor.setAutoDraw(True)
11 message = visual.TextSim(myWin, pos=(0,0), text='Hit Q to quit')
12 trialClock = core.Clock()
13
14 #repeat drawing for each frame
15 while trialClock.getTime() < 20:
16     gabor.setPhase(0.01, '*')
17     message.draw()
18     #handle key presses each frame
19     for keys in event.getKeys(timeStamped=True):
20         if keys[0] in ['escape', 'q']:
21             myWin.close()
22             core.quit()
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

Output

Welcome to PsychoPy2!
v1.63.00

5.1 Builder

When learning a new computer language, the classic first program is simply to print or display “Hello world!”. Lets do it.

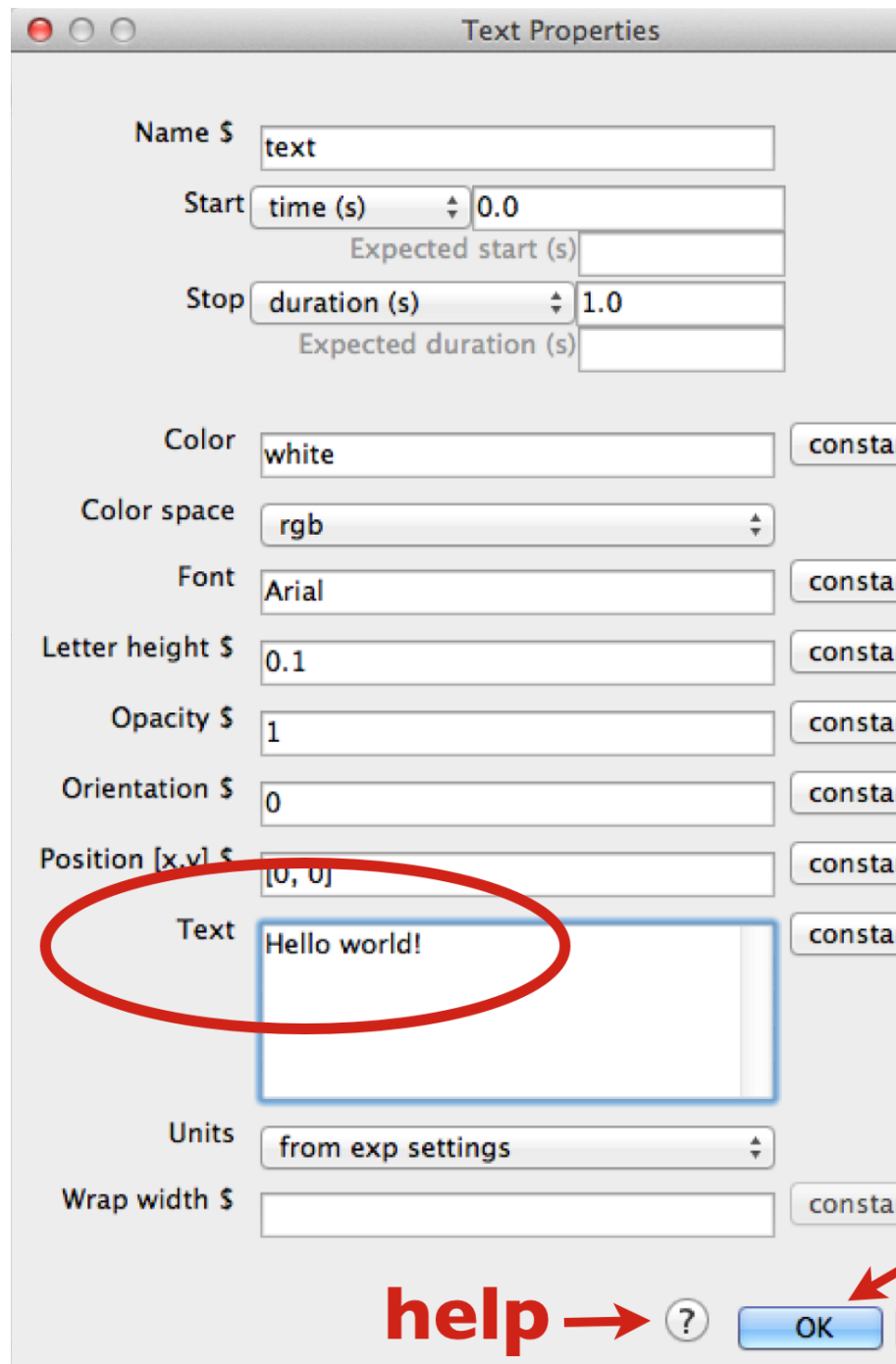
5.1.1 A first program

Start PsychoPy, and be sure to be in the Builder view.

- If you have poked around a bit in the Builder already, be sure to start with a clean slate. To get a new Builder view, type *Ctrl-N* on Windows or Linux, or *Cmd-N* on Mac.



- Click on a Text component

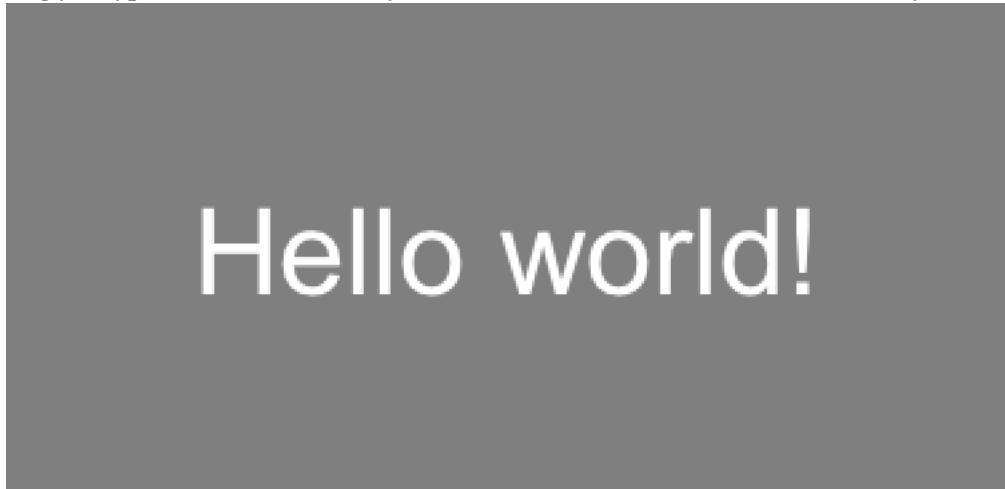


and a Text Properties dialog will pop up.

- In the *Text* field, replace the default text with your message. When you run the program, the text you type here will be shown on the screen.
- Click OK (near the bottom of the dialog box). (Properties dialogs have a link to online help—an icon at the bottom, near the OK button.)
- Your text component now resides in a routine called *trial*. You can click on it to view or edit it. (Components, Routines, and other Builder concepts are explained in the [Builder documentation](#).)
- Back in the main Builder, type *Ctrl-R* (Windows, Linux) or *Cmd-R* (Mac), or use the mouse to click the *Run* icon.



Assuming you typed in “Hello world!”, your screen should have looked like this (briefly):



If nothing happens or it looks wrong, recheck all the steps above; be sure to start from a new Builder view.

What if you wanted to display your cheerful greeting for longer than the default time?

- Click on your Text component (the existing one, not a new one).
- Edit the *Stop duration (s)* to be 3.2; times are in seconds.
- Click OK.
- And finally *Run*.

When running an experiment, you can quit by pressing the *escape* key (this can be configured or disabled). You can quit PsychoPy from the File menu, or typing *Ctrl-Q / Cmd-Q*.

5.1.2 Getting beyond Hello

To do more, you can try things out and see what happens. You may want to consult the [Builder documentation](#). Many people find it helpful to explore the Builder demos, in part to see what is possible, and especially to see how different things are done.

A good way to develop your own first PsychoPy experiment is to base it on the Builder demo that seems closest. Copy it, and then adapt it step by step to become more and more like the program you have in mind. Being familiar with the Builder demos can only help this process.

You could stop here, and just use the Builder for creating your experiments. It provides a lot of the key features that people need to run a wide variety of studies. But it does have its limitations. When you want to have more complex designs or features, you’ll want to investigate the Coder. As a segue to the Coder, let’s start from the Builder, and see how Builder programs work.

5.2 Builder-to-coder

Whenever you run a Builder experiment, PsychoPy will first translate it into python code, and then execute that code.

To get a better feel for what was happening “behind the scenes” in the Builder program above:

- In the Builder, load or recreate your “hello world” program.
- **Instead of running the program, explicitly convert it into python: Type *F5*, or click the *Compile* icon:**



The view will automatically switch to the Coder, and display the python code. If you then save and run this code, it would look the same as running it directly from the Builder.

It is always possible to go from the Builder to python code in this way. You can then edit that code and run it as a python program. However, you cannot go from code back to a Builder representation.

To switch quickly between Builder and Coder views, you can type *Ctrl-L* / *Cmd-L*.

5.3 Coder

Being able to inspect Builder-generated code is nice, but it’s possible to write code yourself, directly. With the Coder and various libraries, you can do virtually anything that your computer is capable of doing, using a full-featured modern programming language (python).

For variety, lets say hello to the Spanish-speaking world. PsychoPy knows Unicode (UTF-8).

If you are not in the Coder, switch to it now.

- Start a new code document: *Ctrl-N* / *Cmd-N*.
- Type (or copy & paste) the following:

```
from psychopy import visual, core

win = visual.Window()
msg = visual.TextStim(win, text=u"\u00A1Hola mundo!")

msg.draw()
win.flip()
core.wait(1)
win.close()
```

- Save the file (the same way as in Builder).
- Run the script.

Note that the same events happen on-screen with this code version, despite the code being much simpler than the code generated by the Builder. (The Builder actually does more, such as prompt for a subject number.)

Coder Shell

The shell provides an interactive python interpreter, which means you can enter commands here to try them out. This provides yet another way to send your salutations to the world. By default, the Coder’s output window is shown at the bottom of the Coder window. Click on the Shell tab, and you should see python’s interactive prompt, `>>>`:

```
PyShell in PsychoPy - type some commands!

Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the prompt, type:

```
>>> print u"\u00A1Hola mundo!"
```

You can do more complex things, such as type in each line from the Coder example directly into the Shell window, doing so line by line:

```
>>> from psychopy import visual, core
```

and then:

```
>>> win = visual.Window()
```

and so on—watch what happens each line::

```
>>> msg = visual.TextStim(win, text=u"\u00A1Hola mundo!")
>>> msg.draw()
>>> win.flip()
```

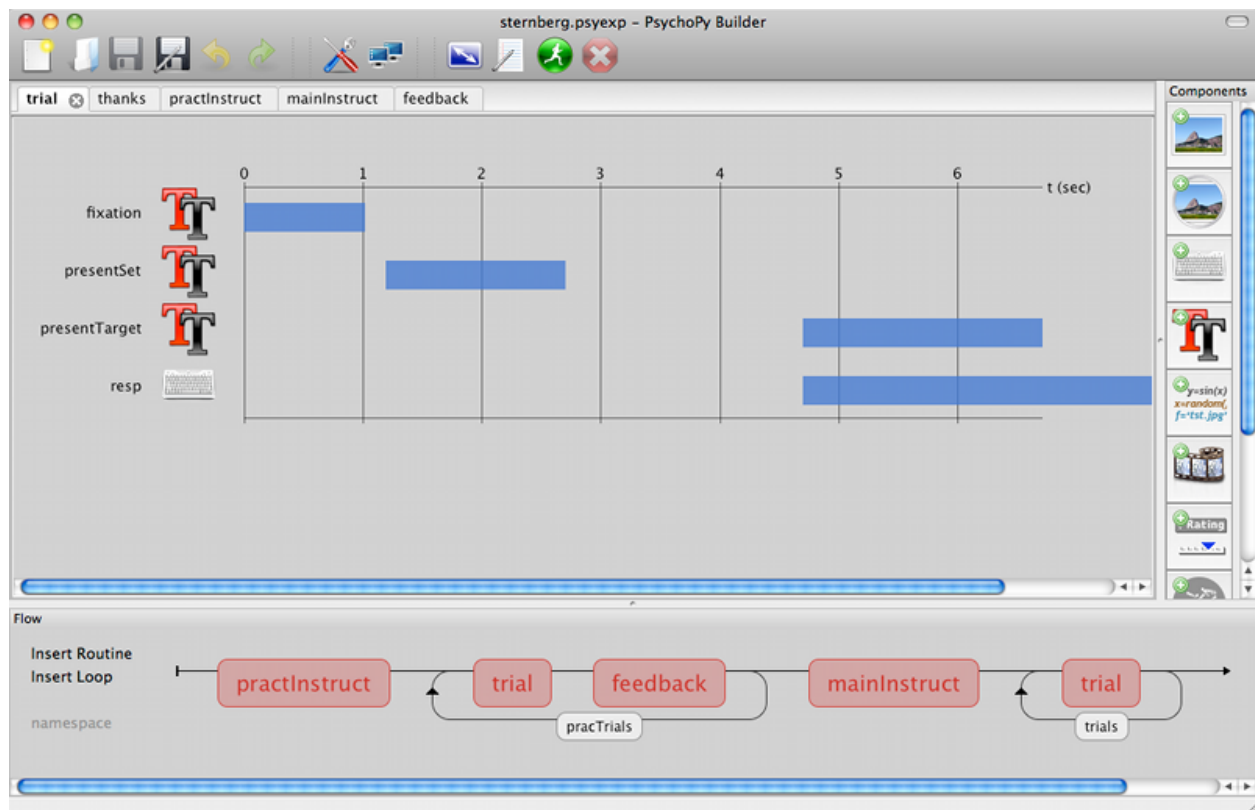
and so on. This lets you try things out and see what happens line-by-line (which is how python goes through your program).

Builder

Building experiments in a GUI

You can now see a [youtube PsychoPy tutorial](#) showing you how to build a simple experiment in the Builder interface

Note: The Builder view is now (at version 1.75) fairly well-developed and should be able to construct a wide variety of studies. But you should still check carefully that the stimuli and response collection are as expected.



Contents:

6.1 Builder concepts

6.1.1 Routines and Flow

The Builder view of the PsychoPy application is designed to allow the rapid development of a wide range of experiments for experimental psychology and cognitive neuroscience experiments.

The Builder view comprises two main panels for viewing the experiment's *Routines* (upper left) and another for viewing the *Flow* (lower part of the window).

An experiment can have any number of *Routines*, describing the timing of stimuli, instructions and responses. These are portrayed in a simple track-based view, similar to that of video-editing software, which allows stimuli to come on go off repeatedly and to overlap with each other.

The way in which these *Routines* are combined and/or repeated is controlled by the *Flow* panel. All experiments have exactly one *Flow*. This takes the form of a standard flowchart allowing a sequence of routines to occur one after another, and for loops to be inserted around one or more of the *Routines*. The loop also controls variables that change between repetitions, such as stimulus attributes.

6.1.2 Example 1 - a reaction time experiment

For a simple reaction time experiment there might be 3 *Routines*, one that presents instructions and waits for a keypress, one that controls the trial timing, and one that thanks the participant at the end. These could then be combined in the *Flow* so that the instructions come first, followed by *trial*, followed by the *thanks Routine*, and a loop could be inserted so that the *Routine* repeated 4 times for each of 6 stimulus intensities.

6.1.3 Example 2 - an fMRI block design

Many fMRI experiments present a sequence of stimuli in a *block*. For this there are multiple ways to create the experiment:

- We could create a single *Routine* that contained a number of stimuli and presented them sequentially, followed by a long blank period to give the inter-epoch interval, and surround this single *Routine* by a loop to control the blocks.
- Alternatively we could create a pair of *Routines* to allow presentation of a) a single stimulus (for 1 sec) and b) a blank screen, for the prolonged period. With these *Routines* we could insert pair of loops, one to repeat the stimulus *Routine* with different images, followed by the blank *Routine*, and another to surround this whole set and control the blocks.

6.1.4 Demos

There are a couple of demos included with the package, that you can find in their own special menu. When you load these the first thing to do is make sure the experiment settings specify the same resolution as your monitor, otherwise the screen can appear off-centred and strangely scaled.

Stroop demo

This runs a digital demonstration of the Stroop effect ¹. The experiment presents a series of coloured words written in coloured 'inks'. Subjects have to report the colour of the letters for each word, but find it harder to do so when the

¹ Stroop, J.R. (1935). "Studies of interference in serial verbal reactions". Journal of Experimental Psychology 18: 643-662.

letters are spelling out a different (incongruous) colour. Reaction times for the congruent trials (where letter colour matches the written word) are faster than for the incongruent trials.

From this demo you should note:

- How to setup a trial list in a .csv or .xlsx file
- How to record key presses and reaction times (using the *resp* Component in *trial Routine*)
- How to change a stimulus parameter on each repetition of the loop. The text and rgb values of the *word* Component are based on *thisTrial*, which represents a single iteration of the *trials* loop. They have been set to change every repeat (don't forget that step!)
- How to present instructions: just have a long-lasting *TextStim* and then force end of the *Routine* when a key is pressed (but don't bother storing the key press).

Psychophysics Staircase demo

This is a mini psychophysics experiment, designed to find the contrast detection threshold of a gabor i.e. find the contrast where the observer can just see the stimulus.

From this demo you should note:

- The opening dialog box requires the participant to enter the orientation of the stimulus, the required fields here are determined by 'Experiment Info' in 'Preferences' which is a python dictionary. This information is then entered into the stimulus parameters using '\$expInfo['ori']'
- The phase of the stimulus is set to change every frame and its value is determined by the value of *trialClock.getTime()*2*. Every *Routine* has a clock associated with it that gets reset at the beginning of the iteration through the *Routine*. There is also a *globalClock* that can be used in the same way. The phase of a *Patch Component* ranges 0-1 (and wraps to that range if beyond it). The result in this case is that the grating drifts at a rate of 2Hz.
- The contrast of the stimulus is determined using an *adaptive staircase*. The *Staircase methods* are different to those used for a loop which uses predetermined values. An important thing to note is that you must define the correct answer.

6.2 Routines

An experiment consists of one or more Routines. A Routine might specify the timing of events within a trial or the presentation of instructions or feedback. Multiple Routines can then be combined in the *Flow*, which controls the order in which these occur and the way in which they repeat.

To create a new Routine, use the Experiment menu. The display size of items within a routine can be adjusted (see the View menu).

Within a Routine there are a number of components. These components determine the occurrence of a stimulus, or the recording of a response. Any number of components can be added to a Routine. Each has its own line in the Routine view that shows when the component starts and finishes in time, and these can overlap.

For now the time axis of the Routines panel is fixed, representing seconds (one line is one second). This will hopefully change in the future so that units can also be number of frames (more precise) and can be scaled up or down to allow very long or very short Routines to be viewed easily. That's on the wishlist...

6.3 Flow

In the Flow panel a number of [Routines](#) can be combined to form an experiment. For instance, your study may have a [Routine](#) that presented initial instructions and waited for a key to be pressed, followed by a [Routine](#) that presented one trial which should be repeated 5 times with various different parameters set. All of this is achieved in the Flow panel. You can adjust the display size of the Flow panel (see View menu).

6.3.1 Adding Routines

The [Routines](#) that the Flow will use should be generated first (although their contents can be added or altered at any time). To insert a [Routine](#) into the Flow click the appropriate button in the left of the Flow panel or use the Experiment menu. A dialog box will appear asking which of your [Routines](#) you wish to add. To select the location move the mouse to the section of the flow where you wish to add it and click on the black disk.

6.3.2 Loops

Loops control the repetition of [Routines](#) and the choice of stimulus parameters for each. PsychoPy can generate the next trial based on the *method of constants* or using an *adaptive staircase*. To insert a loop use the button on the left of the Flow panel, or the item in the Experiment menu of the Builder. The start and end of a loop is set in the same way as the location of a [Routine](#) (see above). Loops can encompass one or more [Routines](#) and other loops (i.e. they can be nested).

As with components in [Routines](#), the loop must be given a name, which must be unique and made up of only alpha-numeric characters (underscores are allowed). I would normally use a plural name, since the loop represents multiple repeats of something. For example, *trials*, *blocks* or *epochs* would be good names for your loops.

It is usually best to use trial information that is contained in an external file (.xlsx or .csv). When inserting a *loop* into the *flow* you can browse to find the file you wish to use for this. An example of this kind of file can be found in the Stroop demo (trialTypes.xlsx). The column names are turned into variables (in this case text, letterColor, corrAns and congruent), these can be used to define parameters in the loop by putting a \$ sign before them e.g. *\$text*.

As the column names from the input file are used in this way they must have legal variable names i.e. they must be unique, have no punctuation or spaces (underscores are ok) and must not start with a digit.

The parameter *Is trials* exists because some loops are not there to indicate trials *per se* but a set of stimuli within a trial, or a set of blocks. In these cases we don't want the data file to add an extra line with each pass around the loop. This parameter can be unchecked to improve (hopefully) your data file outputs. [Added in v1.81.00]

Method of Constants

Selecting a loop type of *random*, *sequential*, or *fullRandom* will result in a *method of constants* experiment, whereby the types of trials that can occur are predetermined. That is, the trials cannot vary depending on how the subject has responded on a previous trial. In this case, a file must be provided that describes the parameters for the repeats. This should be an Excel 2007 (*xlsx*) file or a comma-separated-value (*csv*) file in which columns refer to parameters that are needed to describe stimuli etc. and rows one for each type of trial. These can easily be generated from a spreadsheet package like Excel. (Note that csv files can also be generated using most text editors, as long as they allow you to save the file as "plain text"; other output formats will *not* work, including "rich text".) The top row should be a row of headers: text labels describing the contents of the respective columns. (Headers must also not include spaces or other characters other than letters, numbers or underscores and must not be the same as any variable names used elsewhere in your experiment.) For example, a file containing the following table:

ori	text	corrAns
0	aaa	left
90	aaa	left
0	bbb	right
90	bbb	right

would represent 4 different conditions (or trial types, one per line). The header line describes the parameters in the 3 columns: ori, text and corrAns. It's really useful to include a column called corrAns that shows what the correct key press is going to be for this trial (if there is one).

If the loop type is *sequential* then, on each iteration through the *Routines*, the next row will be selected in the order listed in the file. Under a *random* order, the next row will be selected at random (without replacement); it can only be selected again after all the other rows have also been selected. *nReps* determines how many repeats will be performed (for all conditions). The total number of trials will be the number of conditions (= number of rows in the file, not counting the header row) times the number of repetitions, *nReps*. With the *fullRandom* option, the entire list of trials including repetitions is used in random order, allowing the same item to appear potentially many times in a row, and to repeat without necessarily having done all of the other trials. For example, with 3 repetitions, a file of trial types like this:

letter
a
b
c

could result in the following possible sequences. *sequential* could only ever give one sequence with this order: [a b c a b c a b c]. *random* will give one of 216 different orders (= $3! * 3! * 3! = nReps * (nTrials!)$), for example: [b a c a b c c a b]. Here the letters are effectively in sets of (abc) (abc) (abc), and randomization is only done within each set, ensuring (for example) that there are at least two a's before the subject sees a 3rd b. Finally, *fullRandom* will return one of 362,880 different orders (= $9! = (nReps * nTrials!)$), such as [b b c a a c c a b], which *random* never would. There are no longer mini-blocks or "sets of trials" within the longer run. This means that, by chance, it would also be possible to get a very un-random-looking sequence like [a a a b b b c c c].

It is possible to achieve any sequence you like, subject to any constraints that are logically possible. To do so, in the file you specify every trial in the desired order, and the for the loop select *sequential* order and *nReps*=1.

Selecting a subset of conditions

In the standard *Method of Constants* you would use all the rows/conditions within your conditions file. However there are often times when you want to select a subset of your trials before randomising and repeating.

The parameter *Select rows* allows this. You can specify which rows you want to use by inserting values here:

- 0,2,5 gives the 1st, 3rd and 5th entry of a list - Python starts with index zero)
- *random(4)*10* gives 4 indices from 0 to 10 (so selects 4 out of 11 conditions)
- 5:10 selects the 6th to 9th rows
- *\$myIndices* uses a variable that you've already created

Note in the last case that 5:8 isn't valid syntax for a variable so you cannot do:

myIndices = 5:8

but you can do:

myIndices = slice(5,8) #python object to represent a slice
myIndices = "5:8" #a string that PsychoPy can then parse as a slice later
myIndices = "5:8:2" #as above but

Note that PsychoPy uses Python's built-in slicing syntax (where the first index is zero and the last entry of a slice doesn't get included). You might want to check the outputs of your selection in the Python shell (bottom of the Coder view) like this:

```
>>> range(100)[5:8] #slice 5:8 of a standard set of indices
[5, 6, 7]
>>> range(100)[5:10:2] #slice 5:8 of a standard set of indices
[5, 7, 9, 11, 13, 15, 17, 19]
```

Check that the conditions you wanted to select are the ones you intended!

Staircase methods

The loop type *staircase* allows the implementation of adaptive methods. That is, aspects of a trial can depend on (or “adapt to”) how a subject has responded earlier in the study. This could be, for example, simple up-down staircases where an intensity value is varied trial-by-trial according to certain parameters, or a stop-signal paradigm to assess impulsivity. For this type of loop a ‘correct answer’ must be provided from something like a [Keyboard Component](#). Various parameters for the staircase can be set to govern how many trials will be conducted and how many correct or incorrect answers make the staircase go up or down.

Accessing loop parameters from components

The parameters from your loops are accessible to any component enclosed within that loop. The simplest (and default) way to address these variables is simply to call them by the name of the parameter, prepended with \$ to indicate that this is the name of a variable. For example, if your Flow contains a loop with the above table as its input trial types file then you could give one of your stimuli an orientation *\$ori* which would depend on the current trial type being presented. Example scenarios:

1. You want to loop randomly over some conditions in a loop called *trials*. Your conditions are stored in a csv file with headings ‘ori’, ‘text’, ‘corrAns’ which you provide to this loop. You can then access these values from any component using *\$ori*, *\$text*, and *\$corrAns*
2. You create a random loop called *blocks* and give it an Excel file with a single column called *movieName* listing filenames to be played. On each repeat you can access this with *\$movieName*
3. You create a staircase loop called *stairs*. On each trial you can access the current value in the staircase with *\$thisStair*

Note: When you set a component to use a parameter that will change (e.g on each repeat through the loop) you should **remember to change the component parameter from ‘constant’ to ‘set every repeat’ or ‘set every frame’** or it won't have any effect!

Reducing namespace clutter (advanced)

The downside of the above approach is that the names of trial parameters must be different between every loop, as well as not matching any of the predefined names in python, numpy and PsychoPy. For example, the stimulus called *movie* cannot use a parameter also called *movie* (so you need to call it *movieName*). An alternative method can be used without these restrictions. If you set the Builder preference *unclutteredNamespace* to True you can then access the variables by referring to parameter as an attribute of the singular name of the loop prepended with *this*. For example, if you have a loop called *trials* which has the above file attached to it, then you can access the stimulus ori with *\$thisTrial.ori*. If you have a loop called *blocks* you could use *\$thisBlock.corrAns*.

Now, although the name of the loop must still be valid and unique, the names of the parameters of the file do not have the same requirements (they must still not contain spaces or punctuation characters).

6.4 Components

Routines in the Builder contain any number of components, which typically define the parameters of a stimulus or an input/output device.

The following components are available, as at version 1.65, but further components will be added in the future including Parallel/Serial ports and other visual stimuli (e.g. GeometricStim).

6.4.1 Aperture Component

This component can be used to filter the visual display, as if the subject is looking at it through an opening. Currently only circular apertures are supported. Moreover, only one aperture is enabled at a time. You can't "double up": a second aperture takes precedence.

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the aperture should start having its effect. See *Defining the onset/duration of components* for details.

stop : When the aperture stops having its effect. See *Defining the onset/duration of components* for details.

pos [[X,Y]] The position of the centre of the aperture, in the units specified by the stimulus or window.

size [integer] The size controls how big the aperture will be, in pixels, default = 120

units [pix] What units to use (currently only pix).

See also:

API reference for `Aperture`

6.4.2 Cedrus Button Box Component

This component allows you to connect to a Cedrus Button Box to collect key presses.

Note that there is a limitation currently that a button box can only be used in a single Routine. Otherwise PsychoPy tries to initialise it twice which raises an error. As a workaround, you need to insert the start-routine and each-frame code from the button box into a code component for a second routine.

Properties

Name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the button box is first read. See *Defining the onset/duration of components* for details.

Stop : Governs the duration for which the button box is first read. See *Defining the onset/duration of components* for details.

Force end of Routine [true/false] If this is checked, the first response will end the routine.

Allowed keys [None, or an integer, list, or tuple of integers 0-7] This field lets you specify which buttons (None, or some or all of 0 through 7) to listen to.

Store [(choice of: first, last, all, nothing)] Which button events to save in the data file. Events and the response times are saved, with RT being recorded by the button box (not by PsychoPy).

Store correct [true/false] If selected, a correctness value will be saved in the data file, based on a match with the given correct answer.

Correct answer: button The correct answer, used by Store correct.

Discard previous [true/false] If selected, any previous responses will be ignored (typically this is what you want).

Advanced

Device number: integer This is only needed if you have multiple Cedrus devices connected and you need to specify which to use.

Use box timer [true/false] Set this to True to use the button box timer for timing information (may give better time resolution)

See also:

API reference for `iolab`

6.4.3 Code Component

The *Code Component* can be used to insert short pieces of python code into your experiments. This might be create a variable that you want for another *Component*, to manipulate images before displaying them, to interact with hardware for which there isn't yet a pre-packaged component in *PsychoPy* (e.g. writing code to interact with the serial/parallel ports). See *code uses* below.

Be aware that the code for each of the components in your *Routine* are executed in the order they appear on the *Routine* display (from top to bottom). If you want your *Code Component* to alter a variable to be used by another component immediately, then it needs to be above that component in the view. You may want the code not to take effect until next frame however, in which case put it at the bottom of the *Routine*. You can move *Components* up and down the *Routine* by right-clicking on their icons.

Within your code you can use other variables and modules from the script. For example, all routines have a stopwatch-style `currentTime = trialClock.getTime()`

To see what other variables you might want to use, and also what terms you need to avoid in your chunks of code, *compile your script* before inserting the code object and take a look at the contents of that script.

Note that this page is concerned with *Code Components* specifically, and not all cases in which you might use python syntax within the Builder. It is also possible to put code into a non-code input field (such as the duration or text of a *Text Component*). The syntax there is slightly different (requiring a \$ to trigger the special handling, or \ \$ to avoid triggering special handling). The syntax to use within a Code Component is always regular python syntax.

Parameters

The parameters of the *Code Component* simply specify the code that will get executed at 5 different points within the experiment. You can use as many or as few of these as you need for any *Code Component*:

Begin Experiment: Things that need to be done just once, like importing a supporting module, initialising a variable for later use.

Begin Routine: Certain things might need to be done just once at the start of a *Routine* e.g. at the beginning of each trial you might decide which side a stimulus will appear

Each Frame: Things that need to be updated constantly, throughout the experiment. Note that these will be executed exactly once per video frame (on the order of every 10ms), to give dynamic displays. Static displays do not need to be updated every frame.

End Routine: At the end of the *Routine* (e.g. the trial) you may need to do additional things, like checking if the participant got the right answer

End Experiment: Use this for things like saving data to disk, presenting a graph(?), or resetting hardware to its original state.

Example code uses

1. Set a random location for your target stimulus

There are many ways to do this, but you could add the following to the *Begin Routine* section of a *Code Component* at the top of your *Routine*. Then set your stimulus position to be *\$targetPos* and set the correct answer field of a *Keyboard Component* to be *\$corrAns* (set both of these to update on every repeat of the *Routine*):

```
if random()>0.5:
    targetPos=[-2.0, 0.0]#on the left
    corrAns='left'
else:
    targetPos=[+2.0, 0.0]#on the right
    corrAns='right'
```

2. Create a patch of noise

As with the above there are many different ways to create noise, but a simple method would be to add the following to the *Begin Routine* section of a *Code Component* at the top of your *Routine*. Then set the image as *\$noiseTexture*:

```
noiseTexture = random.rand((128,128)) * 2.0 - 1
```

3. Send a feedback message at the end of the experiment

Make a new routine, and place it at the end of the flow (i.e., the end of the experiment). Create a *Code Component* with this in the *Begin Experiment* field:

```
expClock = core.Clock()
```

and put this in the *Begin routine* field:

```
msg = "Thanks for participating - that took %.2f minutes in total" %(expClock.getTime()) / 60.0)
```

Next, add a *Text Component* to the routine, and set the text to *\$msg*. Be sure that the text field's updating is set to "Set every repeat" (and not "Constant").

4. End a loop early.

Code components can also be used to control the end of a loop. See examples in *Recipes:builderTerminateLoops*.

What variables are available to use?

The most complete way to find this out for your particular script is to *compile it* and take a look at what's in there. Below are some options that appear in nearly all scripts. Remember that those variables are Python objects and can have attributes of their own. You can find out about those attributes using:

```
dir(myObject)
```

Common PsychoPy variables:

- **expInfo**: This is a Python Dictionary containing the information from the starting dialog box. e.g. That generally includes the 'participant' identifier. You can access that in your experiment using `exp['participant']`
- **t**: the current time (in seconds) measured from the start of this Routine
- **frameN**: the number of /completed/ frames since the start of the Routine (=0 in the first frame)
- **win**: the Window that the experiment is using

Your own variables:

- anything you've created in a Code Component is available for the rest of the script. (Sometimes you might need to define it at the beginning of the experiment, so that it will be available throughout.)
- the name of any other stimulus or the parameters from your file also exist as variables.
- most Components have a *status* attribute, which is useful to determine whether a stimulus has *NOT_STARTED*, *STARTED* or *FINISHED*. For example, to play a tone at the end of a Movie Component (of unknown duration) you could set start of your tone to have the 'condition'

```
myMovieName.status==FINISHED
```

Selected contents of the **numpy** library and **numpy.random** are imported by default. The entire numpy library is imported as *np*, so you can use a several hundred maths functions by prepending things with 'np':

- **random()**, **randint()**, **normal()**, **shuffle()** options for creating arrays of random numbers.
- **sin()**, **cos()**, **tan()**, and **pi**: For geometry and trig. By default angles are in radians, if you want the cosine of an angle specified in degrees use `cos(angle*180/pi)`, or use numpy's conversion functions, `rad2deg(angle)` and `deg2rad(angle)`.
- **linspace()**: Create an array of linearly spaced values.
- **log()**, **log10()**: The natural and base-10 log functions, respectively. (It is a lowercase-L in log).
- **sum()**, **len()**: For the sum and length of a list or array. To find an average, it is better to use `average()` (due to the potential for integer division issues with `sum()/len()`).
- **average()**, **sqrt()**, **std()**: For average (mean), square root, and standard deviation, respectively. **Note**: Be sure that the numpy standard deviation formula is the one you want!
- **np._____**: Many math-related features are available through the complete numpy libraries, which are available within psychopy builder scripts as 'np.'. For example, you could use `np.hanning(3)` or `np.random.poisson(10, 10)` in a code component.

6.4.4 Dots (RDK) Component

The Dots Component allows you to present a Random Dot Kinematogram (RDK) to the participant of your study. These are fields of dots that drift in different directions and subjects are typically required to identify the 'global motion' of the field.

There are many ways to define the motion of the signal and noise dots. In PsychoPy the way the dots are configured follows Scase, Braddick & Raymond (1996). Although Scase et al (1996) show that the choice of algorithm for your dots actually makes relatively little difference there are some **potential** gotchas. Think carefully about whether each of these will affect your particular case:

- **limited dot lifetimes:** as your dots drift in one direction they go off the edge of the stimulus and are replaced randomly in the stimulus field. This could lead to a higher density of dots in the direction of motion providing subjects with an alternative cue to direction. Keeping dot lives relatively short prevents this.
- **noiseDots='direction':** some groups have used noise dots that appear in a random location on each frame (noiseDots='location'). This has the disadvantage that the noise dots not only have a random direction but also a random speed (whereas signal dots have a constant speed and constant direction)
- **signalDots='same':** on each frame the dots constituting the signal could be the same as on the previous frame or different. If 'different', participants could follow a single dot for a long time and calculate its average direction of motion to get the 'global' direction, because the dots would sometimes take a random direction and sometimes take the signal direction.

As a result of these, the defaults for PsychoPy are to have signalDots that are from a 'different' population, noise dots that have random 'direction' and a dot life of 3 frames.

Parameters

name : Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

units [None, 'norm', 'cm', 'deg' or 'pix'] If None then the current units of the Window will be used. See *Units for the window and stimuli* for explanation of other options.

nDots [int] number of dots to be generated

fieldPos [(x,y) or [x,y]] specifying the location of the centre of the stimulus.

fieldSize [a single value, specifying the diameter of the field] Sizes can be negative and can extend beyond the window.

fieldShape : Defines the shape of the field in which the dots appear. For a circular field the nDots represents the *average* number of dots per frame, but on each frame this may vary a little.

dotSize Always specified in pixels

dotLife [int] Number of frames each dot lives for (-1=infinite)

dir [float (degrees)] Direction of the signal dots

speed [float] Speed of the dots (in *units* per frame)

signalDots : If 'same' then the signal and noise dots are constant. If different then the choice of which is signal and which is noise gets randomised on each frame. This corresponds to Scase et al's (1996) categories of RDK.

noiseDots ['direction', 'position' or 'walk'] Determines the behaviour of the noise dots, taken directly from Scase et al's (1996) categories. For 'position', noise dots take a random position every frame. For 'direction' noise dots follow a random, but constant direction. For 'walk' noise dots vary their direction every frame, but keep a constant speed.

See also:

API reference for DotStim

6.4.5 Grating Component

The Grating stimulus allows a texture to be wrapped/cycled in 2 dimensions, optionally in conjunction with a mask (e.g. Gaussian window). The texture can be a bitmap image from a variety of standard file formats, or a synthetic texture such as a sinusoidal grating. The mask can also be derived from either an image, or mathematical form such as a Gaussian.

When using gratings, if you want to use the *spatial frequency* setting then create just a single cycle of your texture and allow PsychoPy to handle the repetition of that texture (do not create the cycles you're expecting within the texture).

Gratings can have their position, orientation, size and other settings manipulated on a frame-by-frame basis. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), however this is slight and would not be noticed in the majority of experiments.

Parameters

Name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear. See [Defining the onset/duration of components](#) for details.

Stop : Governs the duration for which the stimulus is presented. See [Defining the onset/duration of components](#) for details.

Color : See [Color spaces](#)

Color space [rgb, dkl or lms] See [Color spaces](#)

Opacity [0-1] Can be used to create semi-transparent gratings

Orientation [degrees] The orientation of the entire patch (texture and mask) in degrees.

Position [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

Size [[size_x, size_y] or a single value (applied to x and y)] The size of the stimulus in the given units of the stimulus/window. If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. $sd = size/6$)

Units [deg, cm, pix, norm, or inherit from window] See [Units for the window and stimuli](#)

Advanced Settings

Texture: a filename, a standard name (sin, sqr) or a variable giving a numpy array This specifies the image that will be used as the *texture* for the visual patch. The image can be repeated on the patch (in either x or y or both) by setting the spatial frequency to be high (or can be stretched so that only a subset of the image appears by setting the spatial frequency to be low). Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

Mask [a filename, a standard name (gauss, circle, raisedCos) or a numpy array of dimensions NxNx1] The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

Interpolate : If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. *Nearest* will use a nearest-neighbour rule.

Phase [single float or pair of values [X,Y]] The position of the texture within the mask, in both X and Y. If a single value is given it will be applied to both dimensions. The phase has units of cycles (rather than degrees or radians), wrapping at 1. As a result, setting the phase to 0,1,2... is equivalent, causing the texture to be centered on the mask. A phase of 0.25 will cause the image to shift by half a cycle (equivalent to pi radians). The advantage of this is that if you set the phase according to time it is automatically in Hz.

Spatial Frequency [[SFx, SFy] or a single value (applied to x and y)] The spatial frequency of the texture on the patch. The units are dependent on the specified units for the stimulus/window; if the units are *deg* then the SF units will be *cycles/deg*, if units are *norm* then the SF units will be cycles per stimulus. If this is set to none then only one cycle will be displayed.

Texture Resolution [an integer (power of two)] Defines the size of the resolution of the texture for standard textures such as *sin*, *sqr* etc. For most cases a value of 256 pixels will suffice, but if stimuli are going to be very small then a lower resolution will use less memory.

See also:

API reference for `GratingStim`

6.4.6 Image Component

The Image stimulus allows an image to be presented, which can be a bitmap image from a variety of standard file formats, with an optional transparency mask that can effectively control the shape of the image. The mask can also be derived from an image file, or mathematical form such as a Gaussian.

It is a really good idea to get your image in roughly the size (in pixels) that it will appear on screen to save memory. If you leave the resolution at 12 megapixel camera, as taken from your camera, but then present it on a standard screen at 1680x1050 (=1.6 megapixels) then PsychoPy and your graphics card have to do an awful lot of unnecessary work. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), but this is slight and would not be noticed in the majority of experiments.

Images can have their position, orientation, size and other settings manipulated on a frame-by-frame basis.

Parameters

Name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear. See [Defining the onset/duration of components](#) for details.

Stop : Governs the duration for which the stimulus is presented. See [Defining the onset/duration of components](#) for details.

Image [a filename or a standard name (*sin*, *sqr*)] Filenames can be relative or absolute paths and can refer to most image formats (e.g. *tif*, *jpg*, *bmp*, *png*, etc.). If this is set to none, the patch will be a flat colour.

Position [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

Size [[size_x, size_y] or a single value (applied to x and y)] The size of the stimulus in the given units of the stimulus/window. If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. *sd=size/6*) Set this to be blank to get the image in its native size.

Orientation [degrees] The orientation of the entire patch (texture and mask) in degrees.

Opacity [value from 0 to 1] If opacity is reduced then the underlying images/stimuli will show through

Units [deg, cm, pix, norm, or inherit from window] See [Units for the window and stimuli](#)

Advanced Settings

Color [Colors can be applied to luminance-only images (not to rgb images)] See [Color spaces](#)

Color space [to be used if a color is supplied] See [Color spaces](#)

Mask [a filename, a standard name (gauss, circle, raisedCos) or a numpy array of dimensions NxNx1] The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

Interpolate : If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. *Nearest* will use a nearest-neighbour rule.

Texture Resolution: This is only needed if you use a synthetic texture (e.g. sinusoidal grating) as the image.

See also:

API reference for `ImageStim`

6.4.7 ioLab Systems buttonbox Component

A button box is a hardware device that is used to collect participant responses with high temporal precision, ideally with true ms accuracy.

Both the response (which button was pressed) and time taken to make it are returned. The time taken is determined by a clock on the device itself. This is what makes it capable (in theory) of high precision timing.

Check the log file to see how long it takes for PsychoPy to reset the button box's internal clock. If this takes a while, then the RT timing values are not likely to be high precision. It might be possible for you to obtain a correction factor for your computer + button box set up, if the timing delay is highly reliable.

The ioLabs button box also has a built-in voice-key, but PsychoPy does not have an interface for it. Use a microphone component instead.

Properties

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See [Defining the onset/duration of components](#) for details.

stop : The duration for which the stimulus is presented. See [Defining the onset/duration of components](#) for details.

Force end of Routine [checkbox] If this is checked, the first response will end the routine.

Active buttons [None, or an integer, list, or tuple of integers 0-7] The ioLabs box lets you specify a set of active buttons. Responses on non-active buttons are ignored by the box, and never sent to PsychoPy. This field lets you specify which buttons (None, or some or all of 0 through 7).

Lights : If selected, the lights above the active buttons will be turned on.

Using code components, it is possible to turn on and off specific lights within a trial. See the API for `iolab`.

Store [(choice of: first, last, all, nothing)] Which button events to save in the data file. Events and the response times are saved, with RT being recorded by the button box (not by PsychoPy).

Store correct [checkbox] If selected, a correctness value will be saved in the data file, based on a match with the given correct answer.

Correct answer: button The correct answer, used by Store correct.

Discard previous [checkbox] If selected, any previous responses will be ignored (typically this is what you want).

Lights off [checkbox] If selected, all lights will be turned off at the end of each routine.

See also:

API reference for `iolab`

6.4.8 Keyboard Component

The Keyboard component can be used to collect responses from a participant.

By not storing the key press and checking the *forceEndTrial* box it can be used simply to end a *Routine*

Parameters

Name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start [float or integer] The time that the keyboard should first get checked. See *Defining the onset/duration of components* for details.

Stop : When the keyboard is no longer checked. See *Defining the onset/duration of components* for details.

Force end routine If this box is checked then the *Routine* will end as soon as one of the *allowed keys* is pressed.

Allowed keys A list of allowed keys can be specified here, e.g. ['m','z','1','2'], or the name of a variable holding such a list. If this box is left blank then any key that is pressed will be read. Only *allowed keys* count as having been pressed; any other key will not be stored and will not force the end of the Routine. Note that key names (even for number keys) should be given in single quotes, separated by commas. Cursor control keys can be accessed with 'up', 'down', and so on; the space bar is 'space'. To find other special keys, run the Coder Input demo, "what_key.py", press the key, and check the Coder output window.

Store Which key press, if any, should be stored; the first to be pressed, the last to be pressed or all that have been pressed. If the key press is to force the end of the trial then this setting is unlikely to be necessary, unless two keys happen to be pressed in the same video frame. The response time will also be stored if a keypress is recorded. This time will be taken from the start of keyboard checking (e.g. if the keyboard was initiated 2 seconds into the trial and a key was pressed 3.2s into the trials the response time will be recorded as 1.2s).

Store correct Check this box if you wish to store whether or not this key press was correct. If so then fill in the next box that defines what would constitute a correct answer e.g. left, 1 or \$corrAns (note this should not be in inverted commas). This is given as Python code that should return True (1) or False (0). Often this correct answer will be defined in the settings of the *Loops*.

Discard previous Check this box to ensure that only key presses that occur during this keyboard checking period are used. If this box is not checked a keyboard press that has occurred before the start of the checking period will be interpreted as the first keyboard press. For most experiments this box should be checked.

See also:

API reference for [psychopy.event](#)

6.4.9 Microphone Component

Please note: This is a new component, and is subject to change.

The microphone component provides a way to record sound during an experiment. To do so, specify the starting time relative to the start of the routine (see *start* below) and a stop time (= duration in seconds). A blank duration evaluates to recording for 0.000s.

The resulting sound files are saved in .wav format (at 48000 Hz, 16 bit), one file per recording. The files appear in a new folder within the data directory (the subdirectory name ends in *_wav*). The file names include the unix (epoch) time of the onset of the recording with milliseconds, e.g., *mic-1346437545.759.wav*.

It is possible to stop a recording that is in progress by using a code component. Every frame, check for a condition (such as key 'q', or a mouse click), and call the *.stop()* method of the microphone component. The recording will

end at that point and be saved. For example, if *mic* is the name of your microphone component, then in the code component, do this on **Each frame**:

```
if event.getKeys(['q']):  
    mic.stop()
```

Parameters

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the stimulus should first play. See *Defining the onset/duration of components* for details.

stop (duration): The length of time (sec) to record for. An *expected duration* can be given for visualisation purposes. See *Defining the onset/duration of components* for details; note that only seconds are allowed.

See also:

API reference for AdvAudioCapture

6.4.10 Mouse Component

The Mouse component can be used to collect responses from a participant. The coordinates of the mouse location are given in the same coordinates as the Window, with (0,0) in the centre.

Scenarios

This can be used in various ways. Here are some scenarios (email the list if you have other uses for your mouse):

Use the mouse to record the location of a button press

Use the mouse to control stimulus parameters Imagine you want to use your mouse to make your ‘patch’_ bigger or smaller and save the final size. Call your *mouse* ‘mouse’, set it to save its state at the end of the trial and set the button press to end the Routine. Then for the size setting of your Patch stimulus insert *\$mouse.getPos()[0]* to use the x position of the mouse to control the size or *\$mouse.getPos()[1]* to use the y position.

Tracking the entire path of the mouse during a period

Parameters

Name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the mouse should first be checked. See *Defining the onset/duration of components* for details.

stop : When the mouse is no longer checked. See *Defining the onset/duration of components* for details.

Force End Routine on Press If this box is checked then the *Routine* will end as soon as one of the mouse buttons is pressed.

Save Mouse State How often do you need to save the state of the mouse? Every time the subject presses a mouse button, at the end of the trial, or every single frame? Note that the text output for cases where you store the mouse data repeatedly per trial (e.g. every press or every frame) is likely to be very hard to interpret, so you may then need to analyse your data using the psydat file (with python code) instead. Hopefully in future releases the output of the text file will be improved.

Time Relative To Whenever the mouse state is saved (e.g. on button press or at end of trial) a time is saved too. Do you want this time to be relative to start of the *Routine*, or the start of the whole experiment?

See also:

API reference for `Mouse`

6.4.11 Movie Component

The Movie component allows movie files to be played from a variety of formats (e.g. mpeg, avi, mov).

The movie can be positioned, rotated, flipped and stretched to any size on the screen (using the [Units for the window and stimuli](#) given).

Parameters

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See [Defining the onset/duration of components](#) for details.

stop : Governs the duration for which the stimulus is presented (if you want to cut a movie short). Usually you can leave this blank and insert the *Expected* duration just for visualisation purposes. See [Defining the onset/duration of components](#) for details.

movie [string] The filename of the movie, including the path. The path can be absolute or relative to the location of the experiment (.psyexp) file.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

ori [degrees] Movies can be rotated in real-time too! This specifies the orientation of the movie in degrees.

size [[size_x, size_y] or a single value (applied to both x and y)] The size of the stimulus in the given units of the stimulus/window.

units [deg, cm, pix, norm, or inherit from window] See [Units for the window and stimuli](#)

See also:

API reference for `MovieStim`

6.4.12 Parallel Port Out Component

This component allows you to send triggers to a parallel port or to a LabJack device.

An example usage would be in EEG experiments to set the port to 0 when no stimuli are present and then set it to an identifier value for each stimulus synchronised to the start/stop of that stimulus. In that case you might set the *Start data* to be *\$ID* (with ID being a column in your conditions file) and set the *Stop Data* to be 0.

Properties

Name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear. See [Defining the onset/duration of components](#) for details.

Stop : Governs the duration for which the stimulus is presented. See [Defining the onset/duration of components](#) for details.

Port address [select the appropriate option] You need to know the address of the parallel port you wish to write to. The options that appear in this drop-down list are determined by the application preferences. You can add your particular port there if you prefer.

Start data [0-255] When the start time/condition occurs this value will be sent to the parallel port. The value is given as a byte (a value from 0-255) controlling the 8 data pins of the parallel port.

Stop data [0-255] As with start data but sent at the end of the period.

Sync to screen [boolean] If true then the parallel port will be sent synchronised to the next screen refresh, which is ideal if it should indicate the onset of a visual stimulus. If set to False then the data will be set on the parallel port immediately.

See also:

API reference for `iolab`

6.4.13 Patch (image) Component

The Patch stimulus allows images to be presented in a variety of forms on the screen. It allows the combination of an image, which can be a bitmap image from a variety of standard file formats, or a synthetic repeating texture such as a sinusoidal grating. A transparency mask can also be control the shape of the image, and this can also be derived from either a second image, or mathematical form such as a Gaussian.

Patches can have their position, orientation, size and other settings manipulated on a frame-by-frame basis. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), however this is slight and would not be noticed in the majority of experiments.

Parameters

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

image [a filename, a standard name ('sin', 'sqr') or a numpy array of dimensions NxNx1 or NxNx3] This specifies the image that will be used as the *texture* for the visual patch. The image can be repeated on the patch (in either x or y or both) by setting the spatial frequency to be high (or can be stretched so that only a subset of the image appears by setting the spatial frequency to be low). Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

mask [a filename, a standard name ('gauss', 'circle') or a numpy array of dimensions NxNx1] The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

ori [degrees] The orientation of the entire patch (texture and mask) in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

size [[size_x, size_y] or a single value (applied to x and y)] The size of the stimulus in the given units of the stimulus/window. If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. $sd = size/6$)

units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

Advanced Settings

colour : See [Color spaces](#)

colour space [rgb, dkl or lms] See [Color spaces](#)

SF [[SFx, SFy] or a single value (applied to x and y)] The spatial frequency of the texture on the patch. The units are dependent on the specified units for the stimulus/window; if the units are *deg* then the SF units will be *cycles/deg*, if units are *norm* then the SF units will be cycles per stimulus. If this is set to none then only one cycle will be displayed.

phase [single float or pair of values [X,Y]] The position of the texture within the mask, in both X and Y. If a single value is given it will be applied to both dimensions. The phase has units of cycles (rather than degrees or radians), wrapping at 1. As a result, setting the phase to 0,1,2... is equivalent, causing the texture to be centered on the mask. A phase of 0.25 will cause the image to shift by half a cycle (equivalent to pi radians). The advantage of this is that if you set the phase according to time it is automatically in Hz.

Texture Resolution [an integer (power of two)] Defines the size of the resolution of the texture for standard textures such as *sin*, *sqr* etc. For most cases a value of 256 pixels will suffice, but if stimuli are going to be very small then a lower resolution will use less memory.

interpolate : If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. *Nearest* will use a nearest-neighbour rule.

See also:

API reference for `PatchStim`

6.4.14 Polygon (shape) Component

(added in version 1.78.00)

The Polygon stimulus allows you to present a wide range of regular geometric shapes. The basic control comes from setting the

- 2 vertices give a line
- 3 give a triangle
- 4 give a rectangle etc.
- a large number will approximate a circle/ellipse

The size parameter takes two values. For a line only the first is used (then use *ori* to specify the orientation). For triangles and rectangles the size specifies the height and width as expected. Note that for pentagons upwards, however, the size determines the width/height of the ellipse on which the vertices will fall, rather than the width/height of the vertices themselves (slightly smaller typically).

Parameters

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

nVertices : integer

The number of vertices for your shape (2 gives a line, 3 gives a triangle,... a large number results in a circle/ellipse). It is not (currently) possible to vary the number of vertices dynamically.

fill settings:

Control the color inside the shape. If you set this to *None* then you will have a transparent shape (the line will remain)

line settings:

Control color and width of the line. The line width is always specified in pixels - it does not honour the *units* parameter.

size [[w,h]] See note above

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

ori [degrees] The orientation of the entire patch (texture and mask) in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

units [deg, cm, pix, norm, or inherit from window] See [Units for the window and stimuli](#)

See also:

API reference for `Polygon` API reference for `Rect` API reference for `ShapeStim` #for arbitrary vertices

6.4.15 RatingScale Component

A rating scale is used to collect a numeric rating or a choice from a few alternatives, via the mouse, the keyboard, or both. Both the response and time taken to make it are returned.

A given routine might involve an image (patch component), along with a rating scale to collect the response. A routine from a personality questionnaire could have text plus a rating scale.

Three common usage styles are enabled on the first settings page: ‘visual analog scale’: the subject uses the mouse to position a marker on an unmarked line

‘category choices’: choose among verbal labels (categories, e.g., “True, False” or “Yes, No, Not sure”)

‘scale description’: used for numeric choices, e.g., 1 to 7 rating

Complete control over the display options is available as an advanced setting, ‘customize_everything’.

Properties

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : The duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

visualAnalogScale [checkbox] If this is checked, a line with no tick marks will be presented using the ‘glow’ marker, and will return a rating from 0.00 to 1.00 (quasi-continuous). This is intended to bias people away from thinking in terms of numbers, and focus more on the visual bar when making their rating. This supersedes either choices or scaleDescription.

category choices [string] Instead of a numeric scale, you can present the subject with words or phrases to choose from. Enter all the words as a string. (Probably more than 6 or so will not look so great on the screen.) Spaces are assumed to separate the words. If there are any commas, the string will be interpreted as a list of words or phrases (possibly including spaces) that are separated by commas.

scaleDescription : Brief instructions, reminding the subject how to interpret the numerical scale, default = “1 = not at all ... extremely = 7”

low [str] The lowest number (bottom end of the scale), default = 1. If it’s not an integer, it will be converted to lowAnchorText (see Advanced).

high [str] The highest number (top end of the scale), default = 7. If it’s not an integer, it will be converted to highAnchorText (see Advanced).

Advanced settings

single click : If this box is checked the participant can only click the scale once and their response will be stored. If this box is not checked the participant must accept their rating before it is stored.

startTime [float or integer] The time (relative to the beginning of this Routine) that the rating scale should first appear.

forceEndTrial : If checked, when the subject makes a rating the routine will be ended.

size [float] The size controls how big the scale will appear on the screen. (Same as “displaySizeFactor”.) Larger than 1 will be larger than the default, smaller than 1 will be smaller than the default.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window. Default is centered left-right, and somewhat lower than the vertical center (0, -0.4).

duration : The maximum duration in seconds for which the stimulus is presented. See duration for details. Typically, the subject’s response should end the trial, not a duration. A blank or negative value means wait for a very long time.

storeRatingTime: Save the time from the beginning of the trial until the participant responds.

storeRating: Save the rating that was selected

lowAnchorText [str] Custom text to display at the low end of the scale, e.g., “0%”; overrides ‘low’ setting

highAnchorText [str] Custom text to display at the low end of the scale, e.g., “100%”; overrides ‘high’ setting

customize_everything [str] If this is not blank, it will be used when initializing the rating scale just as it would be in a code component (see RatingScale). This allows access to all the customizable aspects of a rating scale, and supersedes all of the other RatingScale settings in the dialog panel. (This does not affect: startTime, forceEndTrial, duration, storeRatingTime, storeRating.)

See also:

API reference for RatingScale

6.4.16 Sound Component

Parameters

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the stimulus should first play. See [Defining the onset/duration of components](#) for details.

stop : For sounds loaded from a file leave this blank and then give the *Expected duration* below for visualisation purposes. See [Defining the onset/duration of components](#) for details.

sound : This sound can be described in a variety of ways:

- a number can specify the frequency in Hz (e.g. 440)

- a letter gives a note name (e.g. “C”) and sharp or flat can also be added (e.g. “Csh” “Bf”)
- a filename, which can be a relative or absolute path (mid, wav, and ogg are supported).

volume [float or integer] The volume with which the sound should be played. It’s a normalized value between 0 (minimum) and 1 (maximum).

See also:

API reference for `SoundPyo`

6.4.17 Static Component

(Added in Version 1.78.00)

The Static Component allows you to have a period where you can preload images or perform other time-consuming operations that not be possible while the screen is being updated.

Typically a static period would be something like an inter-trial or inter-stimulus interval (ITI/ISI). During this period you should not have any other objects being presented that are being updated (this isn’t checked for you - you have to make that check yourself), but you can have components being presented that are themselves static. For instance a fixation point never changes and so it can be presented during the static period (it will be presented and left on-screen while the other updates are being made).

Any stimulus updates can be made to occur during any static period defined in the experiment (it does not have to be in the same Routine). This is done in the updates selection box- once a static period exists it will show up here as well as the standard options of *constant* and *every repeat* etc. Many parameter updates (e.g. orientation are made so quickly that using the static period is of no benefit but others, most notably the loading of images from disk, can take substantial periods of time and these should always be performed during a static period to ensure good timing.

If the updates that have been requested were not completed by the end of the static period (i.e. there was a timing overshoot) then you will receive a warning to that effect. In this case you either need a longer static period to perform the actions or you need to reduce the time required for the action (e.g. use an image with fewer pixels).

Parameters

name : Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the static period begins. See [Defining the onset/duration of components](#) for details.

stop : The time that the static period ends. See [Defining the onset/duration of components](#) for details.

custom code : After running the component updates (which are defined in each component, not here) any code inserted here will also be run

See also:

API reference for `StaticPeriod`

6.4.18 Text Component

This component can be used to present text to the participant, either instructions or stimuli.

name [string] Everything in a PsychoPy experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See [Defining the onset/duration of components](#) for details.

stop : The duration for which the stimulus is presented. See [Defining the onset/duration of components](#) for details.

color : See [Color spaces](#)

color space [rgb, dkl or lms] See [Color spaces](#)

ori [degrees] The orientation of the stimulus in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

height [integer or float] The height of the characters in the given units of the stimulus/window. Note that nearly all actual letters will occupy a smaller space than this, depending on font, character, presence of accents etc. The width of the letters is determined by the aspect ratio of the font.

units [deg, cm, pix, norm, or inherit from window] See [Units for the window and stimuli](#)

opacity : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

flip : Whether to mirror-reverse the text: ‘horiz’ for left-right mirroring, ‘vert’ for up-down mirroring. The flip can be set dynamically on a per-frame basis by using a variable, e.g., \$mirror, as defined in a code component or conditions file and set to either ‘horiz’ or ‘vert’.

See also:

API reference for `TextStim`

6.4.19 Entering parameters

Most of the entry boxes for Component parameters simply receive text or numeric values or lists (sequences of values surrounded by square brackets) as input. In addition, the user can insert variables and code into most of these, which will be interpreted either at the beginning of the experiment or at regular intervals within it.

To indicate to PsychoPy that the value represents a variable or python code, rather than literal text, it should be preceded by a \$. For example, inserting *intensity* into the text field of the Text Component will cause that word literally to be presented, whereas *\$intensity* will cause python to search for the variable called intensity in the script.

Variables associated with [Loops](#) can also be entered in this way (see [Accessing loop parameters from components](#) for further details). But it can also be used to evaluate arbitrary python code.

For example:

- `$random(2)` will generate a pair of random numbers
- `$"yn"[randint(2)]` will randomly choose the first or second character (y or n)
- `$globalClock.getTime()` will insert the current time in secs of the globalClock object
- `$(sin(angle), cos(angle))` will insert the sin and cos of an angle (e.g. into the x,y coords of a stimulus)

6.4.20 How often to evaluate the variable/code

If you do want the parameters of a stimulus to be evaluated by code in this way you need also to decide how often it should be updated. By default, the parameters of Components are set to be *constant*; the parameter will be set at the beginning of the experiment and will remain that way for the duration. Alternatively, they can be set to change either on *every repeat* in which case the parameter will be set at the beginning of the Routine on each repeat of it. Lastly many parameters can even be set *on every frame*, allowing them to change constantly on every refresh of the screen.

6.5 Experiment settings

The settings menu can be accessed by clicking the icon at the top of the window. It allows the user to set various aspects of the experiment, such as the size of the window to be used or what information is gathered about the subject and determine what outputs (data files) will be generated.

6.5.1 Settings

Basic settings

Experiment name: A name that will be stored in the metadata of the data file.

Show info dlg: If this box is checked then a dialog will appear at the beginning of the experiment allowing the *Experiment Info* to be changed.

Experiment Info: This information will be presented in a dialog box at the start and will be saved with any data files and so can be used for storing information about the current run of the study. The information stored here can also be used within the experiment. For example, if the *Experiment Info* included a field called *ori* then Builder Components could access `expInfo['ori']` to retrieve the orientation set here. Obviously this is a useful way to run essentially the same experiment, but with different conditions set at run-time.

Enable escape: If ticked then the *Esc* key can be used to exit the experiment at any time (even without a keyboard component)

Data settings

Data filename: (new in version 1.80.00): A *formatted string* to control the base filename and path, often based on variables such as the date and/or the participant. This base filename will be given the various extensions for the different file types as needed. Examples:

```
# all in data folder: data/JWP_memoryTask_2014_Feb_15_1648
'data/%s_%s_%s' %(expInfo['participant'], expName, expInfo['date'])

# group by participant folder: data/JWP/memoryTask-2014_Feb_15_1648
'data/%s/%s-%s' %(expInfo['participant'], expName, expInfo['date'])

# put into dropbox: ~/dropbox/data/memoryTask/JWP-2014_Feb_15_1648
# on Windows you may need to replace ~ with your home directory
'~/dropbox/data/%s/%s-%s' %(expName, expInfo['participant'], expInfo['date'])
```

Save Excel file: If this box is checked an Excel data file (.xlsx) will be stored.

Save csv file: If this box is checked a comma separated variable (.csv) will be stored.

Save psydat file: If this box is checked a *PsychoPy data file (.psydat)* will be stored. This is a Python specific format (.pickle files) which contains more information than .xlsx or .csv files that can be used with data analysis and plotting scripts written in Python. Whilst you may not wish to use this format it is recommended that you always save a copy as it contains a complete record of the experiment at the time of data collection.

Save log file A log file provides a record of what occurred during the experiment in chronological order, including information about any errors or warnings that may have occurred.

Logging level How much detail do you want to be output to the log file, if it is being saved. The lowest level is *error*, which only outputs error messages; *warning* outputs warnings and errors; *info* outputs all info, warnings and errors; *debug* outputs all info that can be logged. This system enables the user to get a great deal of information while generating their experiments, but then reducing this easily to just the critical information needed when

actually running the study. If your experiment is not behaving as you expect it to, this is an excellent place to begin to work out what the problem is.

Screen settings

Monitor The name of the monitor calibration. Must match one of the monitor names from [Monitor Center](#).

Screen: If multiple screens are available (and if the graphics card is *not* an intel integrated graphics chip) then the user can choose which screen they use (e.g. 1 or 2).

Full-screen window: If this box is checked then the experiment window will fill the screen (overriding the window size setting and using the size that the screen is currently set to in the operating system settings).

Window size: The size of the window in pixels, if this is not to be a full-screen window.

Units The default units of the window (see [Units for the window and stimuli](#)). These can be overridden by individual [Components](#).

6.6 Defining the onset/duration of components

As of version 1.70.00, the onset and offset times of stimuli can be defined in several ways.

Start and stop times can be entered in terms of seconds (*time (s)*), by frame number (*frameN*) or in relation to another stimulus (*condition*). *Condition* would be used to make [Components](#) start or stop depending on the status of something else, for example when a sound has finished. Duration can also be varied using a [Code Component](#).

If you need very precise timing (particularly for very brief stimuli for instance) then it is best to control your onset/duration by specifying the number of frames the stimulus will be presented for.

Measuring duration in seconds (or milliseconds) is not very precise because it doesn't take into account the fact that your monitor has a fixed frame rate. For example if the screen has a refresh rate of 60Hz you cannot present your stimulus for 120ms; the frame rate would limit you to 116.7ms (7 frames) or 133.3ms (8 frames). The duration of a frame (in seconds) is simply 1/refresh rate in Hz.

Condition would be used to make [Components](#) start or stop depending on the status of something else, for example when a movie has finished. Duration can also be varied using a code component.

In cases where PsychoPy cannot determine the start/endpoint of your Component (e.g. because it is a variable) you can enter an 'Expected' start/duration. This simply allows components with variable durations to be drawn in the Routine window. If you do not enter the approximate duration it will not be drawn, but this will not affect experimental performance.

For more details of how to achieve good temporal precision see [Timing Issues and synchronisation](#)

6.6.1 Examples

- Use *time(s)* or *frameN* and simply enter numeric values into the start and duration boxes.
- Use *time(s)* or *frameN* and enter a numeric value into the start time and set the duration to a variable name by preceding it with a \$ as described [here](#). Then set *expected time* to see an approximation in your [routine](#)
- Use *condition* to cause the stimulus to start immediately after a movie component called myMovie, by entering `$myMovie.status==FINISHED` into the *start time*.

6.7 Generating outputs (datafiles)

There are 4 main forms of *output* file from PsychoPy:

- Excel 2007 files (.xlsx) see *Excel Data Files* for more details
- text data files (.csv, .tsv, or .txt) see *Delimited Text Files* for more details
- binary data files (.psydat) see *PsychoPy Data Files* for more details
- log files (.log) see *Log Files* for more details

6.8 Common Mistakes (aka Gotcha's)

6.8.1 General Advice

- Python and therefore PsychoPy is CASE SENSITIVE
- To use a dollar sign (\$) for anything other than to indicate a code snippet for example in a *Text Component*, precede it with a backslash \\$ (the backslash won't be printed)
- Have you entered your the settings for your *monitor*? If you are using degrees as a unit of measurement and have not entered your monitor settings, the size of stimuli will not be accurate.
- If your experiment is not behaving in the way that you expect. Have you looked at the *log file*? This can point you in the right direction. Did you know you can change the type of information that is stored in the log file in preferences by changing the *logging level*.
- Have you tried compiling the script and running it. Does this produce a particular error message that points you at a particular problem area? You can also change things in a more detailed way in the coder view and if you are having problems, reading through the script can highlight problems. Reading a compiled script can also help with the creation of a *Code Component*

6.8.2 My stimulus isn't appearing, there's only the grey background

- Have you checked the size of your stimulus? If it is 0.5x0.5 pixels you won't be able to see it!
- Have you checked the position of your stimulus? Is it positioned off the screen?

6.8.3 The loop isn't using my Excel spreadsheet

- Have you remembered to specify the file you want to use when setting up the loop?
- Have you remembered to add the variables proceeded by the \$ symbol to your stimuli?

6.8.4 I just want a plain square, but it's turning into a grating

- If you don't want your stimulus to have a texture, you need Image to be None

6.8.5 The code snippet I've entered doesn't do anything

- Have you remembered to put a \$ symbol at the beginning (this isn't necessary, and should be avoided in a *Code Component*)?
- A dollar sign as the first character of a line indicates to PsychoPy that the rest of the line is code. It does not indicate a variable name (unlike in perl or php). This means that if you are, for example, using variables to determine position, enter `$(x,y)`. The temptation is to use `[$x,$y]`, which will not work.

6.8.6 My stimulus isn't changing as I progress through the loop

- Have you changed the setting for the variable that you want to change to 'change every repeat' (or 'change every frame')?

6.8.7 I'm getting the error message `AttributeError: 'unicode object has no attribute 'XXXX''`

- This type of error is usually caused by a naming conflict. Whilst we have made every attempt to make sure that these conflicts produce a warning message it is possible that they may still occur.
- The most common source of naming conflicts in an external file which has been imported to be used in a loop i.e. `.xlsx`, `.csv`.
- Check to make sure that all of the variable names are unique. There can be no repeated variable names anywhere in your experiment.

6.8.8 The window opens and immediately closes

- Have you checked all of your variable entries are accepted commands e.g. `gauss` but not `Gauss`
- If you compile your experiment and run it from the coder window what does the error message say? Does it point you towards a particular variable which may be incorrectly formatted?

If you are having problems getting the application to run please see [Troubleshooting](#)

6.9 Compiling a Script

If you click the *compile script* icon this will display the script for your experiment in the *Coder* window.

This can be used for debugging experiments, entering small amounts of code and learning a bit about writing scripts amongst other things.

The code is fully commented and so this can be an excellent introduction to writing your own code.

6.10 Set up your monitor properly

It's a really good idea to tell PsychoPy about the set up of your monitor, especially the size in cm and pixels and its distance, so that PsychoPy can present your stimuli in units that will be consistent in another lab with a different set up (e.g. cm or degrees of visual angle).

You should do this in *Monitor Center* which can be opened from Builder by clicking on the icon that shows two monitors. In *Monitor Center* you can create settings for multiple configurations, e.g. different viewing distances or different physical devices and then select the appropriate one by name in your experiments or scripts.

Having set up your monitor settings you should then tell PsychoPy which of your monitor setups to use for this experiment by going to the *Experiment settings* dialog.

6.11 Future developments

The builder view still has a few rough edges, but is hopefully fairly usable. Here are some of the ways I hope it will improve:

- More components. Several of the stimuli and events that PsychoPy can handle don't currently show up as components in the builder view, but they can be added easily (take a look inside the components directory to see how easy it is to create a component).
- Dialogue entry validation. Dialogue boxes currently allow you to type almost anything into their windows. The only current checking is that a name is given to the component and that this is unique. More checking is needed to reduce errors.
- Similar to the above, I hope to add suggested entries to go into dialogs, as a form of help. e.g. on right-clicking an entry box, say for stimulus orientation, a context menu should appear with ideas including numeric values, known local variables (e.g. "thisTrial.rgb", based on the existing loops in the *Flow*) and global variable ideas (e.g. "frameN*360")
- Better code output. I hope that the builder output code will illustrate best practice for precise timing and stimulus presentation (it will probably always take more lines than a man-made script, but it should be at least as precise). At the moment that isn't the case. e.g. The builder should strongly recommend an interval between trials where only static stimuli are drawn (e.g. fixation) and update components for this trial in that interval.

Coder

Note: These do not teach you about Python *per se*, and you are recommended also to learn about that (Python has many excellent tutorials for programmers and non-programmers alike). In particular, dictionaries, lists and numpy arrays are used a great deal in most PsychoPy experiments.

You can learn to use the scripting interface to PsychoPy in several ways, and you should probably follow a combination of them:

- *Basic Concepts*: some of the logic of PsychoPy scripting
- *PsychoPy Tutorials*: walk you through the development of some semi-complete experiments
- demos: in the demos menu of Coder view. Many and varied
- use the *Builder* to *compile a script* and see how it works
- check the *Reference Manual (API)* for further details
- ultimately go into PsychoPy and start examining the source code. It's just regular python!

7.1 Basic Concepts

7.1.1 Presenting Stimuli

Note: Before you start, tell PsychoPy about your monitor(s) using the *Monitor Center*. That way you get to use units (like degrees of visual angle) that will transfer easily to other computers.

Stimulus objects

Python is an 'object-oriented' programming language, meaning that most stimuli in PsychoPy are represented by python objects, with various associated methods and information.

Typically you should create your stimulus once, at the beginning of the script, and then change it as you need to later using `set__()` commands. For instance, create your text and then change its color any time you like:

```
from psychopy import visual, core
win = visual.Window([400,400])
message = visual.TextStim(win, text='hello')
```

```
message.setAutoDraw(True) # automatically draw every frame
win.flip()
core.wait(2.0)
message.setText('world') # change properties of existing stim
win.flip()
core.wait(2.0)
```

Setting stimulus attributes

Stimulus attributes are typically set using either

- a string, which is just some characters (as `message.setText('world')` above)
- a scalar (a number; see below)
- an x,y-pair (two numbers; see below)

x,y-pair: PsychoPy is very flexible in terms of input. You can specify the widely used x,y-pairs using these types:

- A Tuple (x, y) with two elements
- A List [x, y] with two elements
- A numpy array([x, y]) with two elements

However, PsychoPy always converts the x,y-pairs to numpy arrays internally. For example, all three assignments of pos are equivalent here:

```
stim.pos = (0.5, -0.2) # Right and a bit up from the center
print stim.pos # array([0.5, -0.2])

stim.pos = [0.5, -0.2]
print stim.pos # array([0.5, -0.2])

stim.pos = numpy.array([0.5, -0.2])
print stim.pos # array([0.5, -0.2])
```

Choose your favorite :-) However, you can't assign elementwise:

```
stim.pos[1] = 4 # has no effect
```

Scalar: Int or Float.

Mostly, scalars are no-brainers to understand. E.g.:

```
stim.ori = 90 # Rotate stimulus 90 degrees
stim.opacity = 0.8 # Make the stimulus slightly transparent.
```

However, scalars can also be used to assign x,y-pairs. In that case, both x and y get the value of the scalar. E.g.:

```
stim.size = 0.5
print stim.size # array([0.5, 0.5])
```

Operations on attributes: Operations during assignment of attributes are a handy way to smoothly alter the appearance of your stimuli in loops.

Most scalars and x,y-pairs support the basic operations:

```
stim.attribute += value # addition
stim.attribute -= value # subtraction
stim.attribute *= value # multiplication
stim.attribute /= value # division
```

```
stim.attribute %= value # modulus
stim.attribute **= value # power
```

They are easy to use and understand on scalars:

```
stim.ori = 5 # 5.0, set rotation
stim.ori += 3.8 # 8.8, rotate clockwise
stim.ori -= 0.8 # 8.0, rotate counterclockwise
stim.ori /= 2 # 4.0, home in on zero
stim.ori **= 3 # 64.0, exponential increase in rotation
stim.ori %= 10 # 4.0, modulus 10
```

However, they can also be used on x,y-pairs in very flexible ways. Here you can use both scalars and x,y-pairs as operators. In the latter case, the operations are element-wise:

```
stim.size = 5 # array([5.0, 5.0]), set quadratic size
stim.size += 2 # array([7.0, 7.0]), increase size
stim.size /= 2 # array([3.5, 3.5]), downscale size
stim.size += (0.5, 2.5) # array([4.0, 6.0]), a little wider and much taller
stim.size *= (2, 0.25) # array([8.0, 1.5]), upscale horizontal and downscale vertical
```

Operations are not meaningful for strings.

Timing

There are various ways to measure and control timing in PsychoPy:

- using frame refresh periods (most accurate, least obvious)
- checking the time on Clock objects
- using `core.wait()` commands (most obvious, least flexible/accurate)

Using `core.wait()`, as in the above example, is clear and intuitive in your script. But it can't be used while something is changing. For more flexible timing, you could use a `Clock()` object from the `core` module:

```
from psychopy import visual, core

#setup stimulus
win=visual.Window([400,400])
gabor = visual.GratingStim(win, tex='sin', mask='gauss', sf=5, name='gabor')
gabor.setAutoDraw(True) # automatically draw every frame
gabor.autoLog=False#or we'll get many messages about phase change

clock = core.Clock()
#let's draw a stimulus for 2s, drifting for middle 0.5s
while clock.getTime() < 2.0: # clock times are in seconds
    if 0.5 <= clock.getTime() < 1.0:
        gabor.setPhase(0.1, '+') # increment by 10th of cycle
    win.flip()
```

Clocks are accurate to around 1ms (better on some platforms), but using them to time stimuli is not very accurate because it fails to account for the fact that one frame on your monitor has a fixed frame rate. In the above, the stimulus does not actually get drawn for exactly 0.5s (500ms). If the screen is refreshing at 60Hz (16.7ms per frame) and the `getTime()` call reports that the time has reached 1.999s, then the stimulus will draw again for a frame, in accordance with the `while` loop statement and will ultimately be displayed for 2.0167s. Alternatively, if the time has reached 2.001s, there will not be an extra frame drawn. So using this method you get timing accurate to the nearest frame period but with little consistent precision. An error of 16.7ms might be acceptable to long-duration stimuli, but not to a brief presentation. It also might also give the false impression that a stimulus can be presented for any given period.

At 60Hz refresh you can not present your stimulus for, say, 120ms; the frame period would limit you to a period of 116.7ms (7 frames) or 133.3ms (8 frames).

As a result, the most precise way to control stimulus timing is to present them for a specified number of frames. The frame rate is extremely precise, much better than ms-precision. Calls to `Window.flip()` will be synchronised to the frame refresh; the script will not continue until the flip has occurred. As a result, on most cards, as long as frames are not being ‘dropped’ (see [Detecting dropped frames](#)) you can present stimuli for a fixed, reproducible period.

Note: Some graphics cards, such as Intel GMA graphics chips under win32, don’t support frame sync. Avoid integrated graphics for experiment computers wherever possible.

Using the concept of fixed frame periods and `flip()` calls that sync to those periods we can time stimulus presentation extremely precisely with the following:

```
from psychopy import visual, core

#setup stimulus
win=visual.Window([400,400])
gabor = visual.GratingStim(win, tex='sin', mask='gauss', sf=5,
    name='gabor', autoLog=False)
fixation = visual.GratingStim(win, tex=None, mask='gauss', sf=0, size=0.02,
    name='fixation', autoLog=False)

clock = core.Clock()
#let's draw a stimulus for 2s, drifting for middle 0.5s
for frameN in range(200):#for exactly 200 frames
    if 10 <= frameN < 150: # present fixation for a subset of frames
        fixation.draw()
    if 50 <= frameN < 100: # present stim for a different subset
        gabor.setPhase(0.1, '+') # increment by 10th of cycle
        gabor.draw()
    win.flip()
```

Using autoDraw

Stimuli are typically drawn manually on every frame in which they are needed, using the `draw()` function. You can also set any stimulus to start drawing every frame using `setAutoDraw(True)` or `setAutoDraw(False)`. If you use these commands on stimuli that also have `autoLog=True`, then these functions will also generate a log message on the frame when the first drawing occurs and on the first frame when it is confirmed to have ended.

7.1.2 Logging data

TrialHandler and StairHandler can both generate data outputs in which responses are stored, in relation to the stimulus conditions. In addition to those data outputs, PsychoPy can create detailed chronological log files of events during the experiment.

Log levels and targets

Log messages have various levels of severity: ERROR, WARNING, DATA, EXP, INFO and DEBUG

Multiple *targets* can also be created to receive log messages. Each target has a particular critical level and receives all logged messages greater than that. For example, you could set the console (visual output) to receive only warnings

and errors, have a central log file that you use to store warning messages across studies (with file mode *append*), and another to create a detailed log of data and events within a single study with *level=INFO*:

```
from psychopy import logging
logging.console.setLevel(logging.WARNING)
#overwrite (filemode='w') a detailed log of the last run in this dir
lastLog=logging.LogFile("lastRun.log", level=logging.INFO, filemode='w')
#also append warnings to a central log file
centralLog=logging.LogFile("C:\\psychopyExps.log", level=logging.WARNING, filemode='a')
```

Updating the logs

For performance purposes log files are not actually written when the log commands are ‘sent’. They are stored in a list and processed automatically when the script ends. You might also choose to force a *flush* of the logged messages manually during the experiment (e.g. during an inter-trial interval):

```
from psychopy import logging

...

logging.flush() #write messages out to all targets
```

This should only be necessary if you want to see the logged information as the experiment progresses.

AutoLogging

New in version 1.63.00

Certain events will log themselves automatically by default. For instance, visual stimuli send log messages every time one of their parameters is changed, and when autoDraw is toggled they send a message that the stimulus has started/stopped. All such log messages are timestamped with the frame flip on which they take effect. To avoid this logging, for stimuli such as fixation points that might not be critical to your analyses, or for stimuli that change constantly and will flood the logging system with messages, the autoLogging can be turned on/off at initialisation of the stimulus and can be altered afterwards with *.setAutoLog(True/False)*

Manual methods

In addition to a variety of automatic logging messages, you can create your own, of various levels. These can be timestamped immediately:

```
from psychopy import logging
logging.log(level=logging.WARN, msg='something important')
logging.log(level=logging.EXP, msg='something about the conditions')
logging.log(level=logging.DATA, msg='something about a response')
logging.log(level=logging.INFO, msg='something less important')
```

There are additional convenience functions for the above: *logging.warn*(‘a warning’) etc.

For stimulus changes you probably want the log message to be timestamped based on the frame flip (when the stimulus is next presented) rather than the time that the log message is sent:

```
from psychopy import logging, visual
win = visual.Window([400,400])
win.flip()
logging.log(level=logging.EXP, msg='sent immediately')
```

```
win.logOnFlip(level=logging.EXP, msg='sent on actual flip')
win.flip()
```

Using a custom clock for logs

New in version 1.63.00

By default times for log files are reported as seconds after the very beginning of the script (often it takes a few seconds to initialise and import all modules too). You can set the logging system to use any given `core.Clock` object (actually, anything with a `getTime()` method):

```
from psychopy import core, logging
globalClock=core.Clock()
logging.setDefaultClock(globalClock)
```

7.1.3 Handling Trials and Conditions

TrialHandler

This is what underlies the random and sequential loop types in *Builder*, they work using the *method of constants*. The `trialHandler` presents a predetermined list of conditions in either a sequential or random (without replacement) order.

see `TrialHandler` for more details.

StairHandler

This generates the next trial using an *adaptive staircase*. The conditions are not predetermined and are generated based on the participant's responses.

Staircases are predominately used in psychophysics to measure the discrimination and detection thresholds. However they can be used in any experiment which varies a numeric value as a result of a 2 alternative forced choice (2AFC) response.

The `StairHandler` systematically generates numbers based on staircase parameters. These can then be used to define a stimulus parameter e.g. spatial frequency, stimulus presentation duration. If the participant gives the incorrect response the number generated will get larger and if the participant gives the correct response the number will get smaller.

see `StairHandler` for more details

7.2 PsychoPy Tutorials

7.2.1 Tutorial 1: Generating your first stimulus

A tutorial to get you going with your first stimulus display.

Know your monitor

PsychoPy has been designed to handle your screen calibrations for you. It is also designed to operate (if possible) in the final experimental units that you like to use e.g. degrees of visual angle.

In order to do this PsychoPy needs to know a little about your monitor. There is a GUI to help with this (select `MonitorCenter` from the tools menu of PsychoPyIDE or run `...site-packages/monitors/MonitorCenter.py`).

In the MonitorCenter window you can create a new monitor name, insert values that describe your monitor and run calibrations like gamma corrections. For now you can just stick to the `[testMonitor]` but give it correct values for your screen size in number of pixels and width in cm.

Now, when you create a window on your monitor you can give it the name 'testMonitor' and stimuli will know how they should be scaled appropriately.

Your first stimulus

Building stimuli is extremely easy. All you need to do is create a `Window`, then some stimuli. Draw those stimuli, then update the window. PsychoPy has various other useful commands to help with timing too. Here's an example. Type it into a coder window, save it somewhere and press run.

```

1 from psychopy import visual, core # import some libraries from PsychoPy
2
3 #create a window
4 mywin = visual.Window([800,600], monitor="testMonitor", units="deg")
5
6 #create some stimuli
7 grating = visual.GratingStim(win=mywin, mask="circle", size=3, pos=[-4,0], sf=3)
8 fixation = visual.GratingStim(win=mywin, size=0.5, pos=[0,0], sf=0, rgb=-1)
9
10 #draw the stimuli and update the window
11 grating.draw()
12 fixation.draw()
13 mywin.update()
14
15 #pause, so you get a chance to see it!
16 core.wait(5.0)

```

Note: For those new to Python. Did you notice that the grating and the fixation stimuli both call `GratingStim` but have different arguments? One of the nice features about python is that you can select which arguments to set. `GratingStim` has over 15 arguments that can be set, but the others just take on default values if they aren't needed.

That's a bit easy though. Let's make the stimulus move, at least! To do that we need to create a loop where we change the phase (or orientation, or position...) of the stimulus and then redraw. Add this code in place of the drawing code above:

```

for frameN in range(200):
    grating.setPhase(0.05, '+') # advance phase by 0.05 of a cycle
    grating.draw()
    fixation.draw()
    mywin.update()

```

That ran for 200 frames (and then waited 5 seconds as well). Maybe it would be nicer to keep updating until the user hits a key instead. That's easy to add too. In the first line add `event` to the list of modules you'll import. Then replace the line:

```
for frameN in range(200):
```

with the line:

```
while True: #this creates a never-ending loop
```

Then, within the loop (make sure it has the same indentation as the other lines) add the lines:

```
if len(event.getKeys())>0: break
event.clearEvents()
```

the first line counts how many keys have been pressed since the last frame. If more than zero are found then we break out of the never-ending loop. The second line clears the event buffer and should always be called after you've collected the events you want (otherwise it gets full of events that we don't care about like the mouse moving around etc...).

Your finished script should look something like this:

```
1 from psychopy import visual, core, event #import some libraries from PsychoPy
2
3 #create a window
4 mywin = visual.Window([800,600],monitor="testMonitor", units="deg")
5
6 #create some stimuli
7 grating = visual.GratingStim(win=mywin, mask='circle', size=3, pos=[-4,0], sf=3)
8 fixation = visual.GratingStim(win=mywin, size=0.2, pos=[0,0], sf=0, rgb=-1)
9
10 #draw the stimuli and update the window
11 while True: #this creates a never-ending loop
12     grating.setPhase(0.05, '+') #advance phase by 0.05 of a cycle
13     grating.draw()
14     fixation.draw()
15     mywin.flip()
16
17     if len(event.getKeys())>0: break
18     event.clearEvents()
19
20 #cleanup
21 mywin.close()
22 core.quit()
```

There are several more simple scripts like this in the demos menu of the Coder and Builder views and many more to download. If you're feeling like something bigger then go to [Tutorial 2: Measuring a JND using a staircase procedure](#) which will show you how to build an actual experiment.

7.2.2 Tutorial 2: Measuring a JND using a staircase procedure

This tutorial builds an experiment to test your just-noticeable-difference (JND) to orientation, that is it determines the smallest angular deviation that is needed for you to detect that a gabor stimulus isn't vertical (or at some other reference orientation). The method presents a pair of stimuli at once with the observer having to report with a key press whether the left or the right stimulus was at the reference orientation (e.g. vertical).

You can download the [full code here](#). Note that the entire experiment is constructed of less than 100 lines of code, including the initial presentation of a dialogue for parameters, generation and presentation of stimuli, running the trials, saving data and outputting a simple summary analysis for feedback. Not bad, eh?

There are a great many modifications that can be made to this code, however this example is designed to demonstrate how much can be achieved with very simple code. Modifying existing is an excellent way to begin writing your own scripts, for example you may want to try changing the appearance of the text or the stimuli.

Get info from the user

The first lines of code import the necessary libraries. We need lots of the psychopy components for a full experiment, as well as python's time library (to get the current date) and numpy (which handles various numerical/mathematical functions):


```
from psychopy import core, visual, gui, data, event
from psychopy.tools.filetools import fromFile, toFile
```

The `try:...except:...` lines allow us to try and load a parameter file from a previous run of the experiment. If that fails (e.g. because the experiment has never been run) then create a default set of parameters. These are easy to store in a python dictionary that we'll call `expInfo`:

```
try: #try to get a previous parameters file
    expInfo = fromFile('lastParams.pickle')
except: #if not there then use a default set
    expInfo = {'observer': 'jwp', 'refOrientation': 0}
```

The last line adds the current date to whichever method was used.

So having loaded those parameters, let's allow the user to change them in a dialogue box (which we'll call `dlg`). This is the simplest form of dialogue, created directly from the dictionary above. the dialogue will be presented immediately to the user and the script will wait until they hit *OK* or *Cancel*.

If they hit *OK* then `dlg.OK=True`, in which case we'll use the updated values and save them straight to a parameters file (the one we try to load above).

If they hit *Cancel* then we'll simply quit the script and not save the values.

```
#present a dialogue to change params
dlg = gui.DlgFromDict(expInfo, title='simple JND Exp', fixed=['dateStr'])
if dlg.OK:
    toFile('lastParams.pickle', expInfo) #save params to file for next time
else:
```

Setup the information for trials

We'll create a file to which we can output some data as text during each trial (as well as *outputting a binary file* at the end of the experiment). We'll create a filename from the subject+date+`".csv"` (note how easy it is to concatenate strings in python just by 'adding' them). *csv* files can be opened in most spreadsheet packages. Having opened a text file for writing, the last line shows how easy it is to send text to this target document.

```
#make a text file to save data
fileName = expInfo['observer'] + expInfo['dateStr']
dataFile = open(fileName+'.csv', 'w') #a simple text file with 'comma-separated-values'
```

PsychoPy allows us to set up an object to handle the presentation of stimuli in a staircase procedure, the *StairHandler*. This will define the increment of the orientation (i.e. how far it is from the reference orientation). The staircase can be configured in many ways, but we'll set it up to begin with an increment of 20deg (very detectable) and home in on the 80% threshold value. We'll step up our increment every time the subject gets a wrong answer and step down if they get three right answers in a row. The step size will also decrease after every 2 reversals, starting with an 8dB step (large) and going down to 1dB steps (smallish). We'll finish after 50 trials.

```
#create the staircase handler
staircase = data.StairHandler(startVal = 20.0,
                             stepType = 'db', stepSizes=[8,4,4,2,2,1,1],
                             nUp=1, nDown=3, #will home in on the 80% threshold
```

Build your stimuli

Now we need to create a window, some stimuli and timers. We need a *~psychopy.visual.Window* in which to draw our stimuli, a fixation point and two *~psychopy.visual.GratingStim* stimuli (one for the target probe and one as the foil).

We can have as many timers as we like and reset them at any time during the experiment, but I generally use one to measure the time since the experiment started and another that I reset at the beginning of each trial.

```
#create window and stimuli
win = visual.Window([800,600],allowGUI=True, monitor='testMonitor', units='deg')
foil = visual.GratingStim(win, sf=1, size=4, mask='gauss', ori=expInfo['refOrientation'])
target = visual.GratingStim(win, sf=1, size=4, mask='gauss', ori=expInfo['refOrientation'])
fixation = visual.GratingStim(win, color=-1, colorSpace='rgb', tex=None, mask='circle', size=0.2)
#and some handy clocks to keep track of time
globalClock = core.Clock()
```

Once the stimuli are created we should give the subject a message asking if they're ready. The next two lines create a pair of messages, then draw them into the screen and then update the screen to show what we've drawn. Finally we issue the command `event.waitKeys()` which will wait for a keypress before continuing.

```
#display instructions and wait
message1 = visual.TextStim(win, pos=[0,+3],text='Hit a key when ready.')
message2 = visual.TextStim(win, pos=[0,-3],
    text="Then press left or right to identify the %.1f deg probe." %expInfo['refOrientation'])
message1.draw()
message2.draw()
fixation.draw()
win.flip()#to show our newly drawn 'stimuli'
#pause until there's a keypress
```

Control the presentation of the stimuli

OK, so we have everything that we need to run the experiment. The following uses a for-loop that will iterate over trials in the experiment. With each pass through the loop the `staircase` object will provide the new value for the intensity (which we will call `thisIncrement`). We will randomly choose a side to present the target stimulus using `numpy.random.random()`, setting the position of the target to be there and the foil to be on the other side of the fixation point.

```
for thisIncrement in staircase: #will step through the staircase
    #set location of stimuli
    targetSide= random.choice([-1,1]) #will be either +1(right) or -1(left)
    foil.setPos([-5*targetSide, 0])
```

Then set the orientation of the foil to be the reference orientation plus `thisIncrement`, draw all the stimuli (including the fixation point) and update the window.

```
#set orientation of probe
foil.setOri(expInfo['refOrientation'] + thisIncrement)

#draw all stimuli
foil.draw()
target.draw()
fixation.draw()
```

Wait for presentation time of 500ms and then blank the screen (by updating the screen after drawing just the fixation point).

```
core.wait(0.5) #wait 500ms; but use a loop of x frames for more accurate timing in fullscreen
                # eg, to get 30 frames: for f in xrange(30): win.flip()

#blank screen
fixation.draw()
```

Get input from the subject

Still within the for-loop (note the level of indentation is the same) we need to get the response from the subject. The method works by starting off assuming that there hasn't yet been a response and then waiting for a key press. For each key pressed we check if the answer was correct or incorrect and assign the response appropriately, which ends the trial. We always have to clear the event buffer if we're checking for key presses like this

```
#get response
thisResp=None
while thisResp==None:
    allKeys=event.waitKeys()
    for thisKey in allKeys:
        if thisKey=='left':
            if targetSide== -1: thisResp = 1#correct
            else: thisResp = -1          #incorrect
        elif thisKey=='right':
            if targetSide== 1: thisResp = 1#correct
            else: thisResp = -1          #incorrect
        elif thisKey in ['q', 'escape']:
            core.quit() #abort experiment
```

Now we must tell the staircase the result of this trial with its `addData()` method. Then it can work out whether the next trial is an increment or decrement. Also, on each trial (so still within the for-loop) we may as well save the data as a line of text in that .csv file we created earlier.

```
#add the data to the staircase so it can calculate the next level
staircase.addData(thisResp)
dataFile.write('%i,%.3f,%i\n' %(targetSide, thisIncrement, thisResp))
```

Output your data and clean up

OK! We're basically done! We've reached the end of the for-loop (which occurred because the staircase terminated) which means the trials are over. The next step is to close the text data file and also save the staircase as a binary file (by 'pickling' the file in Python speak) which maintains a lot more info than we were saving in the text file.

```
#staircase has ended
dataFile.close()
```

While we're here, it's quite nice to give some immediate feedback to the user. Let's tell them the intensity values at the all the reversals and give them the mean of the last 6. This is an easy way to get an estimate of the threshold, but we might be able to do a better job by trying to reconstruct the psychometric function. To give that a try see the staircase analysis script of [Tutorial 3](#).

Having saved the data you can give your participant some feedback and quit!

```
staircase.saveAsPickle(fileName) #special python binary file to save all the info

#give some output to user in the command line in the output window
print 'reversals:'
print staircase.reversalIntensities
print 'mean of final 6 reversals = %.3f' %(numpy.average(staircase.reversalIntensities[-6:]))

#give some on screen feedback
feedback1 = visual.TextStim(win, pos=[0,+3],
    text='mean of final 6 reversals = %.3f' %
    (numpy.average(staircase.reversalIntensities[-6:])))
feedback1.draw()
```

```
fixation.draw()
win.flip()
event.waitKeys() #wait for participant to respond

win.close()
```

7.2.3 Tutorial 3: Analysing data in Python

You could simply output your data as tab- or comma-separated text files and analyse the data in some spreadsheet package. But the `matplotlib` library in Python also allows for very neat and simple creation of publication-quality plots.

This script shows you how to use a couple of functions from PsychoPy to open some data files (`psychopy.gui.fileOpenDlg()`) and create a psychometric function out of some staircase data (`psychopy.data.functionFromStaircase()`).

Matplotlib is then used to plot the data.

Note: `Matplotlib` and `pylab`. `Matplotlib` is a python library that has similar command syntax to most of the plotting functions in `Matlab(tm)`. It can be imported in different ways; the `import pylab` line at the beginning of the script is the way to import `matplotlib` as well as a variety of other scientific tools (that aren't strictly to do with plotting *per se*).

```
1
2 #This analysis script takes one or more staircase datafiles as input
3 #from a GUI. It then plots the staircases on top of each other on
4 #the left and a combined psychometric function from the same data
5 #on the right
6
7 from psychopy import data, gui, core
8 from psychopy.tools.filetools import fromFile
9 import pylab
10
11 #Open a dialog box to select files from
12 files = gui.fileOpenDlg('.')
13 if not files:
14     core.quit()
15
16 #get the data from all the files
17 allIntensities, allResponses = [],[]
18 for thisFileName in files:
19     thisDat = fromFile(thisFileName)
20     allIntensities.append( thisDat.intensities )
21     allResponses.append( thisDat.data )
22
23 #plot each staircase
24 pylab.subplot(121)
25 colors = 'brgkcmbrgkcm'
26 lines, names = [],[]
27 for fileN, thisStair in enumerate(allIntensities):
28     #lines.extend(pylab.plot(thisStair))
29     #names = files[fileN]
30     pylab.plot(thisStair, label=files[fileN])
31 #pylab.legend()
32
```

```

33 #get combined data
34 combinedInten, combinedResp, combinedN = \
35     data.functionFromStaircase(allIntensities, allResponses, 5)
36 #fit curve - in this case using a Weibull function
37 fit = data.FitFunction('weibullT AFC', combinedInten, combinedResp, \
38     guess=[0.2, 0.5])
39 smoothInt = pylab.arange(min(combinedInten), max(combinedInten), 0.001)
40 smoothResp = fit.eval(smoothInt)
41 thresh = fit.inverse(0.8)
42 print thresh
43
44 #plot curve
45 pylab.subplot(122)
46 pylab.plot(smoothInt, smoothResp, '-')
47 pylab.plot([thresh, thresh], [0, 0.8], '--'); pylab.plot([0, thresh], \
48     [0.8, 0.8], '--')
49 pylab.title('threshold = %0.3f' % (thresh))
50 #plot points
51 pylab.plot(combinedInten, combinedResp, 'o')
52 pylab.ylim([0, 1])
53
54 pylab.show()

```

Reference Manual (API)

Contents:

8.1 `psychopy.core` - basic functions (clocks etc.)

Basic functions, including timing, `rush` (imported), `quit`

`psychopy.core.getTime()`

Get the current time since `psychopy.core` was loaded.

Version Notes: Note that prior to PsychoPy 1.77.00 the behaviour of `getTime()` was platform dependent (on OSX and linux it was equivalent to `psychopy.core.getAbsTime()` whereas on windows it returned time since loading of the module, as now)

`psychopy.core.getAbsTime()`

Return unix time (i.e., whole seconds elapsed since Jan 1, 1970).

This uses the same clock-base as the other timing features, like `getTime()`. The time (in seconds) ignores the time-zone (like `time.time()` on linux). To take the timezone into account, use `int(time.mktime(time.gmtime()))`.

Absolute times in seconds are especially useful to add to generated file names for being unique, informative (= a meaningful time stamp), and because the resulting files will always sort as expected when sorted in chronological, alphabetical, or numerical order, regardless of locale and so on.

Version Notes: This method was added in PsychoPy 1.77.00

`psychopy.core.wait(secs, hogCPUperiod=0.2)`

Wait for a given time period.

If `secs=10` and `hogCPU=0.2` then for 9.8s python's `time.sleep` function will be used, which is not especially precise, but allows the cpu to perform housekeeping. In the final `hogCPUperiod` the more precise method of constantly polling the clock is used for greater precision.

If you want to obtain key-presses during the wait, be sure to use `pyglet` and to `hogCPU` for the entire time, and then call `psychopy.event.getKeys()` after calling `wait()`

If you want to suppress checking for `pyglet` events during the wait, do this once:

```
core.checkPygletDuringWait = False
```

and from then on you can do:: `core.wait(sec)`

This will preserve terminal-window focus during command line usage.

class psychopy.core.Clock

A convenient class to keep track of time in your experiments. You can have as many independent clocks as you like (e.g. one to time responses, one to keep track of stimuli...)

This clock is identical to the *MonotonicClock* except that it can also be reset to 0 or another value at any point.

add (t)

Add more time to the clock's 'start' time (t0).

Note that, by adding time to t0, you make the current time appear less. Can have the effect that `getTime()` returns a negative number that will gradually count back up to zero.

e.g.:

```
timer = core.Clock()
timer.add(5)
while timer.getTime() < 0:
    # do something
```

reset (newT=0.0)

Reset the time on the clock. With no args time will be set to zero. If a float is received this will be the new time on the clock

class psychopy.core.CountdownTimer (start=0)

Similar to a *Clock* except that time counts down from the time of last reset

Typical usage:

```
timer = core.CountdownTimer(5)
while timer.getTime() > 0: # after 5s will become negative
    # do stuff
```

getTime ()

Returns the current time left on this timer in secs (sub-ms precision)

class psychopy.core.MonotonicClock (start_time=None)

A convenient class to keep track of time in your experiments using a sub-millisecond timer.

Unlike the *Clock* this cannot be reset to arbitrary times. For this clock t=0 always represents the time that the clock was created.

Don't confuse this *class* with *core.monotonicClock* which is an *instance* of it that got created when PsychoPy.core was imported. That clock instance is deliberately designed always to return the time since the start of the study.

Version Notes: This class was added in PsychoPy 1.77.00

getLastResetTime ()

Returns the current offset being applied to the high resolution timebase used by Clock.

getTime ()

Returns the current time on this clock in secs (sub-ms precision)

class psychopy.core.StaticPeriod (screenHz=None, win=None, name='StaticPeriod')

A class to help insert a timing period that includes code to be run.

Typical usage:

```
fixation.draw()
win.flip()
ISI = StaticPeriod(screenHz=60)
ISI.start(0.5) # start a period of 0.5s
```



```
stim.image = 'largeFile.bmp' # could take some time
ISI.complete() # finish the 0.5s, taking into account one 60Hz frame

stim.draw()
win.flip() # the period takes into account the next frame flip
# time should now be at exactly 0.5s later than when ISI.start()
# was called
```

Parameters

- **screenHz** – the frame rate of the monitor (leave as None if you don't want this accounted for)
- **win** – if a visual.Window is given then StaticPeriod will also pause/restart frame interval recording
- **name** – give this StaticPeriod a name for more informative logging messages

complete()

Completes the period, using up whatever time is remaining with a call to wait()

Returns 1 for success, 0 for fail (the period overran)

start(duration)

Start the period. If this is called a second time, the timer will be reset and starts again

8.2 psychopy.visual - many visual stimuli

Window to display all stimuli below.

8.2.1 Aperture

8.2.2 BufferImageStim

Attributes

BufferImageStim
BufferImageStim.win
BufferImageStim.buffer
BufferImageStim.rect
BufferImageStim.stim
BufferImageStim.mask
BufferImageStim.units
BufferImageStim.pos
BufferImageStim.ori
BufferImageStim.size
BufferImageStim.contrast
BufferImageStim.color
BufferImageStim.colorSpace
BufferImageStim.opacity
BufferImageStim.interpolate

Continued on next page

Table 8.1 – continued from previous page

BufferImageStim.name
BufferImageStim.autoLog
BufferImageStim.draw
BufferImageStim.autoDraw

Details

8.2.3 Circle

8.2.4 CustomMouse

8.2.5 DotStim

8.2.6 ElementArrayStim

8.2.7 GratingStim

Attributes

GratingStim
GratingStim.win
GratingStim.tex
GratingStim.mask
GratingStim.units
GratingStim.sf
GratingStim.pos
GratingStim.ori
GratingStim.size
GratingStim.contrast
GratingStim.color
GratingStim.colorSpace
GratingStim.opacity
GratingStim.interpolate
GratingStim.texRes
GratingStim.name
GratingStim.autoLog
GratingStim.draw
GratingStim.autoDraw

Details

8.2.8 Helper functions

8.2.9 ImageStim

As of PsychoPy version 1.79.00 *some* of the properties for this stimulus can be set using the syntax:

```
stim.pos = newPos
```

others need to be set with the older syntax:

```
stim.setImage(newImage)
```

Attributes

ImageStim
ImageStim.win
ImageStim.setImage
ImageStim.setMask
ImageStim.units
ImageStim.pos
ImageStim.ori
ImageStim.size
ImageStim.contrast
ImageStim.color
ImageStim.colorSpace
ImageStim.opacity
ImageStim.interpolate
ImageStim.contains
ImageStim.overlaps
ImageStim.name
ImageStim.autoLog
ImageStim.draw
ImageStim.autoDraw
ImageStim.clearTextures

Details

8.2.10 Line

8.2.11 MovieStim

Attributes

MovieStim
MovieStim.win
MovieStim.units
MovieStim.pos
MovieStim.ori
MovieStim.size
MovieStim.opacity
MovieStim.name
MovieStim.autoLog
MovieStim.draw
MovieStim.autoDraw
MovieStim.loadMovie
MovieStim.play
MovieStim.seek

Continued on next page

Table 8.4 – continued from previous page

MovieStim.pause
MovieStim.stop
MovieStim.setFlipHoriz
MovieStim.setFlipVert

Details

8.2.12 PatchStim (deprecated)

8.2.13 Polygon

8.2.14 RadialStim

Attributes

RadialStim
RadialStim.win
RadialStim.tex
RadialStim.mask
RadialStim.units
RadialStim.pos
RadialStim.ori
RadialStim.size
RadialStim.contrast
RadialStim.color
RadialStim.colorSpace
RadialStim.opacity
RadialStim.interpolate
RadialStim.setAngularCycles
RadialStim.setAngularPhase
RadialStim.setRadialCycles
RadialStim.setRadialPhase
RadialStim.name
RadialStim.autoLog
RadialStim.draw
RadialStim.autoDraw
RadialStim.clearTextures

Details

8.2.15 RatingScale

8.2.16 Rect

8.2.17 ShapeStim

Attributes

ShapeStim
ShapeStim.win
ShapeStim.units
ShapeStim.vertices
ShapeStim.closeShape
ShapeStim.pos
ShapeStim.ori
ShapeStim.size
ShapeStim.contrast
ShapeStim.lineColor
ShapeStim.lineColorSpace
ShapeStim.fillColor
ShapeStim.fillColorSpace
ShapeStim.opacity
ShapeStim.interpolate
ShapeStim.name
ShapeStim.autoLog
ShapeStim.draw
ShapeStim.autoDraw

Details

8.2.18 SimpleImageStim

8.2.19 TextBox

Attributes

TextBox

Note: The following `set____()` attributes all have equivalent `get____()` attributes:

TextBox.setText
TextBox.setPosition
TextBox.setOri
TextBox.setHorzAlign
TextBox.setVertAlign
TextBox.setHorzJust
TextBox.setVertJust
TextBox.setFontColor
TextBox.setBorderColor
TextBox.setBackgroundColor
TextBox.setTextGridLineColor
TextBox.setTextGridLineWidth
TextBox.setInterpolated
TextBox.setOpacity
TextBox.setAutoLog

Continued on next page

Table 8.8 – continued from previous page
 TextBox.draw

Note: TextBox also provides the following read-only functions:

TextBox.getSize
TextBox.getName
TextBox.getDisplayedText
TextBox.getValidStrokeWidths
TextBox.getLineSpacing
TextBox.getGlyphPositionForTextIndex
TextBox.getTextGridCellPlacement

Details

8.2.20 TextStim

8.2.21 Window

8.2.22 psychopy.visual.windowframepack - Pack multiple monochrome images into RGB frame

ProjectorFramePacker

8.2.23 psychopy.visual.windowwarp - warping to spherical, cylindrical, or other projections

Warper

Commonly used:

- ImageStim to show images
- TextStim to show text
- TextBox rewrite of TextStim (faster/better but only monospace fonts)

Shapes (all special classes of ShapeStim):

- ShapeStim to draw shapes with arbitrary numbers of vertices
- Rect to show rectangles
- Circle to show circles
- Polygon to show polygons
- Line to show a line

Images and patterns:

- ImageStim to show images
- SimpleImageStim to show images without bells and whistles

- `GratingStim` to show gratings
- `RadialStim` to show annulus, a rotating wedge, a checkerboard etc

Multiple stimuli:

- `ElementArrayStim` to show many stimuli of the same type
- `DotStim` to show and control movement of dots

Other stimuli:

- `MovieStim` to show movies
- `RatingScale` to collect ratings
- `CustomMouse` to change the cursor in windows with GUI. OBS: will be deprecated soon

General purpose (applies to other stimuli):

- `BufferImageStim` to make a faster-to-show “screenshot” of other stimuli
- `Aperture` to restrict visibility area of other stimuli

See also *Helper functions*

8.3 psychopy.data - functions for storing/saving/analysing data

Routines for handling data structures and analysis

8.3.1 ExperimentHandler

```
class psychopy.data.ExperimentHandler(name='', version='', extraInfo=None, runtime-
                                     Info=None, originPath=None, savePickle=True,
                                     saveWideText=True, dataFileName='', autoLog=True)
```

A container class for keeping track of multiple loops/handlers

Useful for generating a single data file from an experiment with many different loops (e.g. interleaved staircases or loops within loops)

Usage `exp = data.ExperimentHandler(name="Face Preference",version='0.1.0')`

Parameters

name [a string or unicode] As a useful identifier later

version [usually a string (e.g. '1.1.0')] To keep track of which version of the experiment was run

extraInfo [a dictionary] Containing useful information about this run (e.g. {'participant':'jwp','gender':'m','orientation':90})

runtimeInfo [`psychopy.info.RunTimeInfo`] Containining information about the system as detected at runtime

originPath [string or unicode] The path and filename of the originating script/experiment If not provided this will be determined as the path of the calling script.

dataFileName [string] This is defined in advance and the file will be saved at any point that the handler is removed or discarded (unless `.abort()` had been called in advance). The handler will attempt to populate the file even in the event of a (not too serious) crash!

savePickle : True (default) or False

saveWideText : True (default) or False

autoLog : True (default) or False

abort ()

Inform the ExperimentHandler that the run was aborted.

Experiment handler will attempt automatically to save data (even in the event of a crash if possible). So if you quit your script early you may want to tell the Handler not to save out the data files for this run. This is the method that allows you to do that.

addData (name, value)

Add the data with a given name to the current experiment.

Typically the user does not need to use this function; if you added your data to the loop and had already added the loop to the experiment then the loop will automatically inform the experiment that it has received data.

Multiple data name/value pairs can be added to any given entry of the data file and is considered part of the same entry until the nextEntry() call is made.

e.g.:

```
# add some data for this trial
exp.addData('resp.rt', 0.8)
exp.addData('resp.key', 'k')
# end of trial - move to next line in data output
exp.nextEntry()
```

addLoop (loopHandler)

Add a loop such as a *TrialHandler* or *StairHandler* Data from this loop will be included in the resulting data files.

loopEnded (loopHandler)

Informs the experiment handler that the loop is finished and not to include its values in further entries of the experiment.

This method is called by the loop itself if it ends its iterations, so is not typically needed by the user.

nextEntry ()

Calling nextEntry indicates to the ExperimentHandler that the current trial has ended and so further addData() calls correspond to the next trial.

saveAsPickle (fileName, fileCollisionMethod='rename')

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters fileCollisionMethod: Collision method passed to handleFileCollision()

saveAsWideText (fileName, delim=None, matrixOnly=False, appendFile=False, encoding='utf-8', fileCollisionMethod='rename')

Saves a long, wide-format text file, with one line representing the attributes and data for a single trial. Suitable for analysis in R and SPSS.

If *appendFile=True* then the data will be added to the bottom of an existing file. Otherwise, if the file exists already it will be overwritten

If *matrixOnly=True* then the file will not contain a header row, which can be handy if you want to append data to an existing file of the same format.

encoding: The encoding to use when saving a the file. Defaults to *utf-8*.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

8.3.2 TrialHandler

```
class psychopy.data.TrialHandler(trialList, nReps, method='random', dataTypes=None,
                                extraInfo=None, seed=None, originPath=None, name='',
                                autoLog=True)
```

Class to handle trial sequencing and data storage.

Calls to `.next()` will fetch the next trial object given to this handler, according to the method specified (random, sequential, fullRandom). Calls will raise a `StopIteration` error if trials have finished.

See `demo_trialHandler.py`

The psydat file format is literally just a pickled copy of the `TrialHandler` object that saved it. You can open it with:

```
from psychopy.tools.filetools import fromFile
dat = fromFile(path)
```

Then you'll find that `dat` has the following attributes that

Parameters

trialList: a simple list (or flat array) of dictionaries specifying conditions. This can be imported from an excel/csv file using `importConditions()`

nReps: number of repeats for all conditions

method: *'random'*, *'sequential'*, or *'fullRandom'* *'sequential'* obviously presents the conditions in the order they appear in the list. *'random'* will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. *'fullRandom'* fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.

dataTypes: (optional) list of names for data storage. e.g. `['corr','rt','resp']`. If not provided then these will be created as needed during calls to `addData()`

extraInfo: A dictionary This will be stored alongside the data and usually describes the experiment and subject ID, date etc.

seed: an integer If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint

originPath: a string describing the location of the script / experiment file path. The psydat file format will store a copy of the experiment if possible. If `originPath==None` is provided here then the `TrialHandler` will still store a copy of the script where it was created. If `OriginPath==-1` then nothing will be stored.

Attributes (after creation)

.data - a dictionary of numpy arrays, one for each data type stored

.trialList - the original list of dicts, specifying the conditions

.thisIndex - the index of the current trial in the original conditions list

.nTotal - the total number of trials that will be run

.nRemaining - the total number of trials remaining

.thisN - total trials completed so far

.thisRepN - which repeat you are currently on
 .thisTrialN - which trial number *within* that repeat
.thisTrial - a dictionary giving the parameters of the current trial
 .finished - True/False for have we finished yet
 .extraInfo - the dictionary of extra info as given at beginning
.origin - the contents of the script or builder experiment that created the handler

addData (*thisType, value, position=None*)

Add data for the current trial

getEarlierTrial (*n=-1*)

Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns 'None' if trying to access a trial prior to the first.

getExp ()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getFutureTrial (*n=1*)

Returns the condition for n trials into the future, without advancing the trials. A negative n returns a previous (past) trial. Returns 'None' if attempting to go beyond the last trial.

getOriginPathAndFile (*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

next ()

Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN and thisIndex. If the trials have ended this method will raise a StopIteration error. This can be handled with code such as:

```
trials = data.TrialHandler(.....)
for eachTrial in trials: # automatically stops when done
    # do stuff
```

or:

```
trials = data.TrialHandler(.....)
while True: # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration: # we got a StopIteration error
        break #break out of the forever loop
    # do stuff here for the trial
```

printAsText (*stimOut=None, dataOut=('all_mean', 'all_std', 'all_raw'), delim='\t', matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

saveAsExcel (*fileName, sheetName='rawData', stimOut=None, dataOut=('n', 'all_mean', 'all_std', 'all_raw'), matrixOnly=False, appendFile=True, fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see [TrialHandler.saveAsText\(\)](#)) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment

and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

Parameters

fileName: string the name of the file to create or append. Can include relative or absolute path

sheetName: string the name of the worksheet within the file

stimOut: list of strings the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to `trialList` parameter of the `TrialHandler` and give here the names of strings specifying entries in that dictionary

dataOut: list of strings specifying the `dataType` and the analysis to be performed, in the form *dataType_analysis*. The data can be any of the types that you added using `trialHandler.data.add()` and the analysis can be either 'raw' or most things in the numpy library, including 'mean', 'std', 'median', 'max', 'min'. e.g. *rt_max* will give a column of max reaction times across the trials assuming that *rt* values have been stored. The default values will output the raw, mean and std of all datatypes found.

appendFile: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string Collision method passed to `handleFileCollision()`
This is ignored if `append` is True.

saveAsPickle (*fileName*, *fileCollisionMethod*='rename')

Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters `fileCollisionMethod`: Collision method passed to `handleFileCollision()`

saveAsText (*fileName*, *stimOut*=None, *dataOut*=('n', 'all_mean', 'all_std', 'all_raw'), *delim*=None, *matrixOnly*=False, *appendFile*=True, *summarised*=True, *fileCollisionMethod*='rename', *encoding*='utf-8')

Write a text file with the data and various chosen stimulus attributes

Parameters

fileName: will have *.tsv* appended and can include path info.

stimOut: the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings

dataOut: a list of strings specifying the `dataType` and the analysis to be performed, in the form *dataType_analysis*. The data can be any of the types that you added using `trialHandler.data.add()` and the analysis can be either 'raw' or most things in the numpy library, including; 'mean', 'std', 'median', 'max', 'min'... The default values will output the raw, mean and std of all datatypes found

delim: allows the user to use a delimiter other than tab (",") is popular with file extension ".csv")

matrixOnly: outputs the data with no header row or extraInfo attached

appendFile: will add this output to the end of the specified file if it already exists

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to *utf-8*.

saveAsWideText (*fileName*, *delim=None*, *matrixOnly=False*, *appendFile=True*, *encoding='utf-8'*, *fileCollisionMethod='rename'*)

Write a text file with the session, stimulus, and data values from each trial in chronological order. Also, return a pandas DataFrame containing same information as the file.

That is, unlike ‘saveAsText’ and ‘saveAsExcel’:

- each row comprises information from only a single trial.
- no summarizing is done (such as collapsing to produce mean and standard deviation values across trials).

This ‘wide’ format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the trialHandler’s ‘extraInfo’ exists, then each entry in there occurs in every row. In builder, this will include any entries in the ‘Experiment info’ field of the ‘Experiment settings’ dialog. In Coder, this information can be set using something like:

```
myTrialHandler.extraInfo = {'SubjID': 'Joan Smith',
                             'Group': 'Control'}
```

Parameters

fileName: if extension is not specified, ‘.csv’ will be appended if the delimiter is ‘,’, else ‘.tsv’ will be appended. Can include path info.

delim: allows the user to use a delimiter other than the default tab (“,” is popular with file extension “.csv”)

matrixOnly: outputs the data with no header row.

appendFile: will add this output to the end of the specified file if it already exists.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to *utf-8*.

setExp (*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

8.3.3 StairHandler

class `psychopy.data.StairHandler` (*startVal*, *nReversals=None*, *stepSizes=4*, *nTrials=0*, *nUp=1*, *nDown=3*, *extraInfo=None*, *method='2AFC'*, *stepType='db'*, *minVal=None*, *maxVal=None*, *originPath=None*, *name=''*, *autoLog=True*, ***kwargs*)

Class to handle smoothly the selection of the next trial and report current values etc. Calls to `next()` will fetch the next object given to this handler, according to the method specified.

See Demos >> ExperimentalControl >> JND_staircase_exp.py

The staircase will terminate when *nTrials* AND *nReversals* have been exceeded. If *stepSizes* was an array and has been exceeded before *nTrials* is exceeded then the staircase will continue to reverse.

nUp and *nDown* are always considered as 1 until the first reversal is reached. The values entered as arguments are then used.

Parameters

startVal: The initial value for the staircase.

nReversals: The minimum number of reversals permitted. If *stepSizes* is a list, but the minimum number of reversals to perform, *nReversals*, is less than the length of this list, PsychoPy will automatically increase the minimum number of reversals and emit a warning.

stepSizes: The size of steps as a single value or a list (or array). For a single value the step size is fixed. For an array or list the step size will progress to the next entry at each reversal.

nTrials: The minimum number of trials to be conducted. If the staircase has not reached the required number of reversals then it will continue.

nUp: The number of 'incorrect' (or 0) responses before the staircase level increases.

nDown: The number of 'correct' (or 1) responses before the staircase level decreases.

extraInfo: A dictionary (typically) that will be stored along with collected data using *saveAsPickle()* or *saveAsText()* methods.

stepType: specifies whether each step will be a jump of the given size in 'db', 'log' or 'lin' units ('lin' means this intensity will be added/subtracted)

method: Not used and may be deprecated in future releases.

stepType: 'db', 'lin', 'log' The type of steps that should be taken each time. 'lin' will simply add or subtract that amount each step, 'db' and 'log' will step by a certain number of decibels or log units (note that this will prevent your value ever reaching zero or less)

minVal: None, or a number The smallest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

maxVal: None, or a number The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

Additional keyword arguments will be ignored.

Notes

The additional keyword arguments ***kwargs* might for example be passed by the *MultiStairHandler*, which expects a *label* keyword for each staircase. These parameters are to be ignored by the *StairHandler*.

addData (*result, intensity=None*)

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

.addResponse(result, intensity) *.addOtherData('dataName', value')*

addOtherData (*dataName, value*)

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

addResponse (*result, intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial

This is essential to advance the staircase to a new intensity level!

Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

calculateNextIntensity ()

Based on current intensity, counter of correct responses, and current direction.

getExp ()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getOriginPathAndFile (originPath=None)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

next ()

Advances to next trial and returns it. Updates attributes; *thisTrial*, *thisTrialN* and *thisIndex*.

If the trials have ended, calling this method will raise a StopIteration error. This can be handled with code such as:

```
staircase = data.StairHandler(.....)
for eachTrial in staircase: # automatically stops when done
    # do stuff
```

or:

```
staircase = data.StairHandler(.....)
while True: # ie forever
    try:
        thisTrial = staircase.next()
    except StopIteration: # we got a StopIteration error
        break # break out of the forever loop
    # do stuff here for the trial
```

printAsText (stimOut=None, dataOut=('all_mean', 'all_std', 'all_raw'), delim='\t', matrixOnly=False)

Exactly like saveAsText() except that the output goes to the screen instead of a file

saveAsExcel (fileName, sheetName='data', matrixOnly=False, appendFile=True, fileCollision-Method='rename')

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see [TrialHandler.saveAsText\(\)](#)) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string the name of the file to create or append. Can include relative or absolute path

sheetName: string the name of the worksheet within the file

matrixOnly: True or False If set to True then only the data itself will be output (no additional info)

appendFile: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string Collision method passed to `handleFileCollision()`
This is ignored if `append` is True.

saveAsPickle (*fileName*, *fileCollisionMethod*='rename')

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters *fileCollisionMethod*: Collision method passed to `handleFileCollision()`

saveAsText (*fileName*, *delim*=None, *matrixOnly*=False, *fileCollisionMethod*='rename',
encoding='utf-8')

Write a text file with the data

Parameters

fileName: a string The name of the file, including path if needed. The extension `.tsv` will be added if not included.

delim: a string the delimiter to be used (e.g. `' '` for tab-delimited, `' '` for csv files)

matrixOnly: True/False If True, prevents the output of the *extraInfo* provided at initialisation.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to `utf-8`.

setExp (*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

8.3.4 MultiStairHandler

class `psychopy.data.MultiStairHandler` (*stairType*='simple', *method*='random', *conditions*=None, *nTrials*=50, *originPath*=None, *name*='',
autoLog=True)

A Handler to allow easy interleaved staircase procedures (simple or QUEST).

Parameters for the staircases, as used by the relevant *StairHandler* or *QuestHandler* (e.g. the *startVal*, *minVal*, *maxVal*...) should be specified in the *conditions* list and may vary between each staircase. In particular, the conditions /must/ include the a *startVal* (because this is a required argument to the above handlers) a *label* to tag the staircase and a *startValSd* (only for QUEST staircases). Any parameters not specified in the conditions file will revert to the default for that individual handler.

If you need to custom the behaviour further you may want to look at the recipe on *Coder - interleave staircases*.

Params

stairType: 'simple' or 'quest' Use a *StairHandler* or *QuestHandler*

method: 'random' or 'sequential' The stairs are shuffled in each repeat but not randomised more than that (so you can't have 3 repeats of the same staircase in a row unless it's the only one still running)

conditions: a list of dictionaries specifying conditions Can be used to control parameters for the different staircases. Can be imported from an Excel file using *psychoPy.data.importConditions* MUST include keys providing, 'startVal', 'label' and 'start-ValSd' (QUEST only). The 'label' will be used in data file saving so should be unique. See Example Usage below.

nTrials=50 Minimum trials to run (but may take more if the staircase hasn't also met its minimal reversals. See *StairHandler*

Example usage:

```
conditions=[
    {'label':'low', 'startVal': 0.1, 'ori':45},
    {'label':'high','startVal': 0.8, 'ori':45},
    {'label':'low', 'startVal': 0.1, 'ori':90},
    {'label':'high','startVal': 0.8, 'ori':90},
]
stairs = data.MultiStairHandler(conditions=conditions, nTrials=50)

for thisIntensity, thisCondition in stairs:
    thisOri = thisCondition['ori']

    # do something with thisIntensity and thisOri

    stairs.addResponse(correctIncorrect) # this is ESSENTIAL

# save data as multiple formats
stairs.saveDataAsExcel(fileName) # easy to browse
stairs.saveAsPickle(fileName) # contains more info
```

addData (*result, intensity=None*)

Deprecated 1.79.00: It was ambiguous whether you were adding the response (0 or 1) or some other data concerning the trial so there is now a pair of explicit methods:

addResponse(corr,intensity) #some data that alters the next trial value

addOtherData('RT', reactionTime) #some other data that won't control staircase

addOtherData (*name, value*)

Add some data about the current trial that will not be used to control the staircase(s) such as reaction time data

addResponse (*result, intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial

This is essential to advance the staircase to a new intensity level!

getExp ()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getOriginPathAndFile (*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

next ()

Advances to next trial and returns it.

This can be handled with code such as:


```
staircase = data.MultiStairHandler(.....)
for eachTrial in staircase: # automatically stops when done
    # do stuff here for the trial
```

or:

```
staircase = data.MultiStairHandler(.....)
while True: # ie forever
    try:
        thisTrial = staircase.next()
    except StopIteration: # we got a StopIteration error
        break # break out of the forever loop
    # do stuff here for the trial
```

printAsText (*delim='\t', matrixOnly=False*)

Write the data to the standard output stream

Parameters

delim: a string the delimiter to be used (e.g. ‘\t’ for tab-delimited, ‘,’ for csv files)

matrixOnly: True/False If True, prevents the output of the *extraInfo* provided at initialisation.

saveAsExcel (*fileName, matrixOnly=False, appendFile=False, fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see *TrialHandler.saveAsText()*) that the data from each staircase will be save in the same file, with the sheet name coming from the ‘label’ given in the dictionary of conditions during initialisation of the Handler.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level (‘intensity’) at each reversal, a list of reversal indices (trial numbers), the raw staircase/intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string the name of the file to create or append. Can include relative or absolute path

matrixOnly: True or False If set to True then only the data itself will be output (no additional info)

appendFile: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string Collision method passed to *handleFileCollision()* This is ignored if append is True.

saveAsPickle (*fileName, fileCollisionMethod='rename'*)

Saves a copy of self (with data) to a pickle file.

This can be reloaded later and further analyses carried out.

Parameters fileCollisionMethod: Collision method passed to *handleFileCollision()*

saveAsText (*fileName*, *delim=None*, *matrixOnly=False*, *fileCollisionMethod='rename'*, *encoding='utf-8'*)

Write out text files with the data.

For MultiStairHandler this will output one file for each staircase that was run, with *_label* added to the *fileName* that you specify above (label comes from the condition dictionary you specified when you created the Handler).

Parameters

fileName: a string The name of the file, including path if needed. The extension *.tsv* will be added if not included.

delim: a string the delimiter to be used (e.g. ' ' for tab-delimited, ',' for csv files)

matrixOnly: True/False If True, prevents the output of the *extraInfo* provided at initialisation.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to *utf-8*.

setExp (*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

8.3.5 QuestHandler

class psychopy.data.QuestHandler (*startVal*, *startValSd*, *pThreshold=0.82*, *nTrials=None*, *stopInterval=None*, *method='quantile'*, *beta=3.5*, *delta=0.01*, *gamma=0.5*, *grain=0.01*, *range=None*, *extraInfo=None*, *minVal=None*, *maxVal=None*, *staircase=None*, *originPath=None*, *name=''*, *autoLog=True*, ***kwargs*)

Class that implements the Quest algorithm for quick measurement of psychophysical thresholds.

Uses Andrew Straw's **QUEST**, which is a Python port of Denis Pelli's Matlab code.

Measures threshold using a Weibull psychometric function. Currently, it is not possible to use a different psychometric function.

Threshold 't' is measured on an abstract 'intensity' scale, which usually corresponds to log10 contrast.

The Weibull psychometric function:

$$p2 = \frac{1}{1 + 10^{-(\beta(x - x_{\text{Threshold}})^\gamma)}}$$

Example:

```
# setup display/window
...
# create stimulus
stimulus = visual.RadialStim(win=win, tex='sinXsin', size=1,
                             pos=[0,0], units='deg')
...
# create staircase object
# trying to find out the point where subject's response is 50 / 50
# if wanted to do a 2AFC then the defaults for pThreshold and gamma
```

```
# are good
staircase = data.QuestHandler(staircase._nextIntensity, 0.2,
                             pThreshold=0.63, gamma=0.01,
                             nTrials=20, minVal=0, maxVal=1)

...
while thisContrast in staircase:
    # setup stimulus
    stimulus.setContrast(thisContrast)
    stimulus.draw()
    win.flip()
    core.wait(0.5)
    # get response
    ...
    # inform QUEST of the response, needed to calculate next level
    staircase.addResponse(thisResp)
    ...
# can now access 1 of 3 suggested threshold levels
staircase.mean()
staircase.mode()
staircase.quantile() # gets the median
```

Typical values for pThreshold are:

- 0.82 which is equivalent to a 3 up 1 down standard staircase
- **0.63 which is equivalent to a 1 up 1 down standard staircase** (and might want gamma=0.01)

The variable(s) nTrials and/or stopSd must be specified.

beta, *delta*, and *gamma* are the parameters of the Weibull psychometric function.

Parameters

startVal: Prior threshold estimate or your initial guess threshold.

startValSd: Standard deviation of your starting guess threshold. Be generous with the sd as QUEST will have trouble finding the true threshold if it's more than one sd from your initial guess.

pThreshold Your threshold criterion expressed as probability of response==1. An intensity offset is introduced into the psychometric function so that the threshold (i.e., the midpoint of the table) yields pThreshold.

nTrials: *None* or a number The maximum number of trials to be conducted.

stopInterval: *None* or a number The minimum 5-95% confidence interval required in the threshold estimate before stopping. If both this and nTrials is specified, whichever happens first will determine when Quest will stop.

method: *'quantile'*, *'mean'*, *'mode'* The method used to determine the next threshold to test. If you want to get a specific threshold level at the end of your staircasing, please use the quantile, mean, and mode methods directly.

beta: 3.5 or a number Controls the steepness of the psychometric function.

delta: 0.01 or a number The fraction of trials on which the observer presses blindly.

gamma: 0.5 or a number The fraction of trials that will generate response 1 when intensity=-Inf.

grain: 0.01 or a number The quantization of the internal table.

range: *None*, or a number The intensity difference between the largest and smallest intensity that the internal table can store. This interval will be centered on the initial guess `tGuess`. QUEST assumes that intensities outside of this range have zero prior probability (i.e., they are impossible).

extraInfo: A dictionary (typically) that will be stored along with collected data using `saveAsPickle()` or `saveAsText()` methods.

minVal: *None*, or a number The smallest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

maxVal: *None*, or a number The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

staircase: *None* or *StairHandler* Can supply a staircase object with intensities and results. Might be useful to give the quest algorithm more information if you have it. You can also call the `importData` function directly.

Additional keyword arguments will be ignored.

Notes

The additional keyword arguments `**kwargs` might for example be passed by the *MultiStairHandler*, which expects a *label* keyword for each staircase. These parameters are to be ignored by the *StairHandler*.

addData (*result*, *intensity=None*)

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

`.addResponse(result, intensity)` `.addOtherData('dataName', value')`

addOtherData (*dataName*, *value*)

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

addResponse (*result*, *intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial

Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

calculateNextIntensity ()

based on current intensity and counter of correct responses

confInterval (*getDifference=False*)

give the range of the 5-95% confidence interval

getExp ()

Return the ExperimentHandler that this handler is attached to, if any. Returns *None* if not attached

getOriginPathAndFile (*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If *originPath* is provided (e.g. from Builder) then this is used otherwise the calling script is the *originPath* (fine from a standard python script).

importData (*intensities*, *results*)

import some data which wasn't previously given to the quest algorithm

incTrials (*nNewTrials*)

increase maximum number of trials Updates attribute: *nTrials*

mean ()

mean of Quest posterior pdf

mode()
mode of Quest posterior pdf

next()
Advances to next trial and returns it. Updates attributes; *thisTrial*, *thisTrialN*, *thisIndex*, *finished*, *intensities*

If the trials have ended, calling this method will raise a `StopIteration` error. This can be handled with code such as:

```
staircase = data.QuestHandler(.....)
for eachTrial in staircase: # automatically stops when done
    # do stuff
```

or:

```
staircase = data.QuestHandler(.....)
while True: # i.e. forever
    try:
        thisTrial = staircase.next()
    except StopIteration: # we got a StopIteration error
        break # break out of the forever loop
    # do stuff here for the trial
```

printAsText (*stimOut=None*, *dataOut=('all_mean', 'all_std', 'all_raw')*, *delim='\t'*, *matrixOnly=False*)

Exactly like `saveAsText()` except that the output goes to the screen instead of a file

quantile (*p=None*)
quantile of Quest posterior pdf

saveAsExcel (*fileName*, *sheetName='data'*, *matrixOnly=False*, *appendFile=True*, *fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string the name of the file to create or append. Can include relative or absolute path

sheetName: string the name of the worksheet within the file

matrixOnly: True or False If set to True then only the data itself will be output (no additional info)

appendFile: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string Collision method passed to `handleFileCollision()`
This is ignored if `append` is True.

saveAsPickle (*fileName*, *fileCollisionMethod*=*'rename'*)

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters *fileCollisionMethod*: Collision method passed to `handleFileCollision()`

saveAsText (*fileName*, *delim*=*None*, *matrixOnly*=*False*, *fileCollisionMethod*=*'rename'*,
encoding=*'utf-8'*)

Write a text file with the data

Parameters

fileName: a string The name of the file, including path if needed. The extension *.tsv* will be added if not included.

delim: a string the delimiter to be used (e.g. *' '* for tab-delimited, *' '* for csv files)

matrixOnly: True/False If True, prevents the output of the *extraInfo* provided at initialisation.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to *utf-8*.

sd()

standard deviation of Quest posterior pdf

setExp (*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

simulate (*tActual*)

returns a simulated user response to the next intensity level presented by Quest, need to supply the actual threshold level

8.3.6 FitWeibull

class `psychopy.data.FitWeibull` (*xx*, *yy*, *sems*=*1.0*, *guess*=*None*, *display*=*1*, *expectedMin*=*0.5*)

Fit a Weibull function (either 2AFC or YN) of the form:

```
y = chance + (1.0-chance)*(1-exp(-(xx/alpha)**(beta)))
```

and with inverse:

```
x = alpha * (-log((1.0-y)/(1-chance)))**(1.0/beta)
```

After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with `[alpha, beta]`)

eval (*xx*, *params*=*None*)

Evaluate *xx* for the current parameters of the model, or for arbitrary params if these are given.

inverse (*yy*, *params*=*None*)

Evaluate *yy* for the current parameters of the model, or for arbitrary params if these are given.

8.3.7 FitLogistic

class psychopy.data.**FitLogistic** (xx, yy, sems=1.0, guess=None, display=1, expectedMin=0.5)
Fit a Logistic function (either 2AFC or YN) of the form:

$$y = \text{chance} + (1 - \text{chance}) / (1 + \exp((\text{PSE} - \text{xx}) * \text{JND}))$$

and with inverse:

$$x = \text{PSE} - \log((1 - \text{chance}) / (\text{yy} - \text{chance}) - 1) / \text{JND}$$

After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with [PSE, JND])

eval (xx, params=None)

Evaluate xx for the current parameters of the model, or for arbitrary params if these are given.

inverse (yy, params=None)

Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.

8.3.8 FitNakaRushton

class psychopy.data.**FitNakaRushton** (xx, yy, sems=1.0, guess=None, display=1, expectedMin=0.5)
Fit a Naka-Rushton function of the form:

$$\text{yy} = \text{rMin} + (\text{rMax} - \text{rMin}) * \text{xx}^n / (\text{xx}^n + \text{c50}^n)$$

After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with [rMin, rMax, c50, n])

Note that this differs from most of the other functions in not using a value for the expected minimum. Rather, it fits this as one of the parameters of the model.

eval (xx, params=None)

Evaluate xx for the current parameters of the model, or for arbitrary params if these are given.

inverse (yy, params=None)

Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.

8.3.9 FitCumNormal

class psychopy.data.**FitCumNormal** (xx, yy, sems=1.0, guess=None, display=1, expectedMin=0.5)
Fit a Cumulative Normal function (aka error function or erf) of the form:

$$y = \text{chance} + (1 - \text{chance}) * ((\text{special.erf}((\text{xx} - \text{xShift}) / (\text{sqrt}(2) * \text{sd})) + 1) * 0.5)$$

and with inverse:

$$x = \text{xShift} + \text{sqrt}(2) * \text{sd} * (\text{erfinv}(((\text{yy} - \text{chance}) / (1 - \text{chance}) - 0.5) * 2))$$

After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with [centre, sd] for the Gaussian distribution forming the cumulative)

NB: Prior to version 1.74 the parameters had different meaning, relating to xShift and slope of the function (similar to 1/sd). Although that is more in with the parameters for the Weibull fit, for instance, it is less in

keeping with standard expectations of normal (Gaussian distributions) so in version 1.74.00 the parameters became the [centre,sd] of the normal distribution.

eval (xx, params=None)

Evaluate xx for the current parameters of the model, or for arbitrary params if these are given.

inverse (yy, params=None)

Evaluate yy for the current parameters of the model, or for arbitrary params if these are given.

8.3.10 importConditions()

psychopy.data.**importConditions** (fileName, returnFieldNames=False, selection='')

Imports a list of conditions from an .xlsx, .csv, or .pkl file

The output is suitable as an input to *TrialHandler* *trialTypes* or to *MultiStairHandler* as a *conditions* list.

If *fileName* ends with:

- .csv: import as a comma-separated-value file** (header + row x col)
- .xlsx: import as Excel 2007 (xlsx) files.** No support for older (.xls) is planned.
- .pkl: import from a pickle file as list of lists** (header + row x col)

The file should contain one row per type of trial needed and one column for each parameter that defines the trial type. The first row should give parameter names, which should:

- be unique
- begin with a letter (upper or lower case)
- contain no spaces or other punctuation (underscores are permitted)

selection is used to select a subset of condition indices to be used It can be a list/array of indices, a python *slice* object or a string to be parsed as either option. e.g.:

- “1,2,4” or [1,2,4] or (1,2,4) are the same
- “2:5” # 2, 3, 4 (doesn’t include last whole value)
- “-10:2:” # tenth from last to the last in steps of 2
- slice(-10, 2, None) # the same as above
- random(5) * 8 # five random vals 0-8

8.3.11 functionFromStaircase()

psychopy.data.**functionFromStaircase** (intensities, responses, bins=10)

Create a psychometric function by binning data from a staircase procedure. Although the default is 10 bins Jon now always uses ‘unique’ bins (fewer bins looks pretty but leads to errors in slope estimation)

usage:

```
intensity, meanCorrect, n = functionFromStaircase(intensities,
                                                  responses, bins)
```

where:

intensities are a list (or array) of intensities to be binned

responses are a list of 0,1 each corresponding to the equivalent intensity value

bins can be an integer (giving that number of bins) or 'unique' (each bin is made from aa data for exactly one intensity value)

intensity a numpy array of intensity values (where each is the center of an intensity bin)

meanCorrect a numpy array of mean % correct in each bin

n a numpy array of number of responses contributing to each mean

8.3.12 bootStraps ()

`psychopy.data.bootStraps (dat, n=1)`

Create a list of n bootstrapped resamples of the data

SLOW IMPLEMENTATION (Python for-loop)

Usage: `out = bootStraps(dat, n=1)`

Where:

dat an NxM or 1xN array (each row is a different condition, each column is a different trial)

n number of bootstrapped resamples to create

out

- `dim[0]`=conditions
- `dim[1]`=trials
- `dim[2]`=resamples

8.4 Encryption

Some labs may wish to better protect their data from casual inspection or accidental disclosure. This is possible within PsychoPy using a separate python package, pyFileSec, which grew out of PsychoPy. pyFileSec is distributed with the StandAlone versions of PsychoPy, or can be installed using pip or easy_install via <https://pypi.python.org/pypi/PyFileSec/>

Some elaboration of pyFileSec usage and security strategy can be found here: <http://pythonhosted.org/PyFileSec>

Basic usage is illustrated in the Coder demo > misc > encrypt_data.py

8.5 `psychopy.event` - for keypresses and mouse clicks

8.6 `psychopy.filters` - helper functions for creating filters

8.7 `psychopy.gui` - create dialogue boxes

8.7.1 `DlgFromDict`

8.7.2 `Dlg`

8.7.3 `fileOpenDlg()`

8.7.4 `fileSaveDlg()`

8.8 `psychopy.hardware` - hardware interfaces

PsychoPy can access a wide range of external hardware. For some devices the interface has already been created in the following sub-packages of PsychoPy. For others you may need to write the code to access the serial port etc. manually.

Contents:

8.8.1 Cedrus (response boxes)

The `pyxid` package, written by Cedrus, is included in the Standalone PsychoPy distributions. See <https://github.com/cedrus-opensource/pyxid> for further info.

Example usage:

```
import pyxid

# get a list of all attached XID devices
devices = pyxid.get_xid_devices()

dev = devices[0] # get the first device to use
if dev.is_response_device():
    dev.reset_base_timer()
    dev.reset_rt_timer()

while True:
    dev.poll_for_response()
    if dev.response_queue_size() > 0:
        response = dev.get_next_response()
        # do something with the response
```

Useful functions

Device classes

8.8.2 Cambridge Research Systems Ltd.

For stimulus display

BitsPlusPlus

Control a CRS Bits# device. See typical usage in the class summary (and in the menu demos>hardware>BitsBox of PsychoPy's Coder view).

Important: See note on *BitsPlusPlusIdentityLUT*

BitsPlusPlus
BitsPlusPlus.mode
BitsPlusPlus.setContrast
BitsPlusPlus.setGamma
BitsPlusPlus.setLUT

Attributes

Details

Finding the identity LUT

For the Bits++ (and related) devices to work correctly it is essential that the graphics card is not altering in any way the values being passed to the monitor (e.g. by gamma correcting). It turns out that finding the 'identity' LUT, where exactly the same values come out as were put in, is not trivial. The obvious LUT would have something like 0/255, 1/255, 2/255... in entry locations 0,1,2... but unfortunately most graphics cards on most operating systems are 'broken' in one way or another, with rounding errors and incorrect start points etc.

PsychoPy provides a few of the common variants of LUT and that can be chosen when you initialise the device using the parameter *rampType*. If no *rampType* is specified then PsychoPy will choose one for you:

```
from psychopy import visual
from psychopy.hardware import crs

win = visual.Window([1024,768], useFBO=True) #we need to be rendering to framebuffer
bits = crs.BitsPlusPlus(win, mode = 'bits++', rampType = 1)
```

The Bits# is capable of reporting back the pixels in a line and this can be used to test that a particular LUT is indeed providing identity values. If you have previously connected a BitsSharp device and used it with PsychoPy then a file will have been stored with a LUT that has been tested with that device. In this case set *rampType* = "configFile" for PsychoPy to use it if such a file is found.

BitsSharp

Control a CRS Bits# device. See typical usage in the class summary (and in the menu demos>hardware>BitsBox of PsychoPy's Coder view).

BitsSharp
BitsSharp.mode
BitsSharp.isAwake
BitsSharp.getInfo
BitsSharp.checkConfig
BitsSharp.gammaCorrectFile
BitsSharp.temporalDithering
BitsSharp.monitorEDID
BitsSharp.beep
BitsSharp.getVideoLine
BitsSharp.start
BitsSharp.stop

Attributes Direct communications with the serial port:

BitsSharp.sendMessage
BitsSharp.getResponse

Control the CLUT (Bits++ mode only):

BitsSharp.setContrast
BitsSharp.setGamma
BitsSharp.setLUT

Details

For display calibration

ColorCAL

ColorCAL

Attributes

Details

8.8.3 egi (pynetstation)

Interface to [EGI Netstation](#)

This is currently a simple import of [pynetstation](#) That needs to be installed (but is included in the *Standalone* distributions of PsychoPy as of version 1.62.01).

installation:

Download the package from the link above and copy egi.py into your site-packages directory.

usage:

```
from psychopy.hardware import egi
```

For an example see the demos menu of the PsychoPy Coder For further documentation see the pynetstation website

8.8.4 Launch an fMRI experiment: Test or Scan

8.8.5 fORP response box

8.8.6 iolab

8.8.7 joystick (pyglet and pygame)

8.8.8 labjacks (USB I/O devices)

PsychoPy provides an interface to the labjack U3 class with a couple of minor additions.

This is accessible by:

```
from psychopy.hardware.labjacks import U3
```

Except for the additional *setdata* function the U3 class operates exactly as that in the U3 library that labjack provides, documented here:

<http://labjack.com/support/labjackpython>

Note: To use labjack devices you do need also to install the driver software described on the page above

8.8.9 Minolta

Minolta light-measuring devices See <http://www.konicaminolta.com/instruments>

```
class psychopy.hardware.minolta.LS100 (port, maxAttempts=1)
```

A class to define a Minolta LS100 (or LS110?) photometer

You need to connect a LS100 to the serial (RS232) port and **when you turn it on press the F key** on the device. This will put it into the correct mode to communicate with the serial port.

usage:

```
from psychopy.hardware import minolta
phot = minolta.LS100(port)
if phot.OK: # then we successfully made a connection
    print(phot.getLum())
```

Parameters port: string

the serial port that should be checked

maxAttempts: int If the device doesn't respond first time how many attempts should be made?

If you're certain that this is the correct port and the device is on and correctly configured then this could be set high. If not then set this low.

Troubleshooting Various messages are printed to the log regarding the function of this device, but to see them you need to set the printing of the log to the correct level:

```
from psychopy import logging
logging.console.setLevel(logging.ERROR)  # error messages only
logging.console.setLevel(logging.INFO)  # more info
logging.console.setLevel(logging.DEBUG)  # log all communications
```

If you're using a keyspan adapter (at least on OS X) be aware that it needs a driver installed. Otherwise no ports will be found.

Error messages:

ERROR: Couldn't connect to Minolta LS100/110 on ____: This likely means that the device is not connected to that port (although the port has been found and opened). Check that the device has the / in the bottom right of the display; if not turn off and on again holding the *F* key.

ERROR: No reply from LS100: The port was found, the connection was made and an initial command worked, but then the device stopped communicating. If the first measurement taken with the device after connecting does not yield a reasonable intensity the device can sulk (not a technical term!). The "[" on the display will disappear and you can no longer communicate with the device. Turn it off and on again (with *F* depressed) and use a reasonably bright screen for your first measurement. Subsequent measurements can be dark (or we really would be in trouble!!).

checkOK (*msg*)

Check that the message from the photometer is OK. If there's an error show it (printed).

Then return True (OK) or False.

clearMemory ()

Clear the memory of the device from previous measurements

getLum ()

Makes a measurement and returns the luminance value

measure ()

Measure the current luminance and set .lastLum to this value

sendMessage (*message, timeout=5.0*)

Send a command to the photometer and wait an allotted timeout for a response.

setMaxAttempts (*maxAttempts*)

Changes the number of attempts to send a message and read the output. Typically this should be low initially, if you aren't sure that the device is setup correctly but then, after the first successful reading, set it higher.

setMode (*mode='04'*)

Set the mode for measurements. Returns True (success) or False

'04' means absolute measurements. '08' = peak '09' = cont

See user manual for other modes

8.8.10 PhotoResearch

Supported devices:

- [PR650](#)

- [PR655/PR670](#)

PhotoResearch spectrophotometers See <http://www.photoresearch.com/>

class `psychopy.hardware.pr.PR650` (*port, verbose=None*)
An interface to the PR650 via the serial port.

(Added in version 1.63.02)

example usage:

```
from psychopy.hardware.pr import PR650
myPR650 = PR650(port)
myPR650.getLum() # make a measurement
nm, power = myPR650.getLastSpectrum() # get a power spectrum for the
last measurement
```

NB `psychopy.hardware.findPhotometer()` will locate and return any supported device for you so you can also do:

```
from psychopy import hardware
phot = hardware.findPhotometer()
print(phot.getLum())
```

Troubleshooting Various messages are printed to the log regarding the function of this device, but to see them you need to set the printing of the log to the correct level:

```
from psychopy import logging
logging.console.setLevel(logging.ERROR) # error messages only
logging.console.setLevel(logging.INFO) # will give more info
logging.console.setLevel(logging.DEBUG) # log all communications
```

If you're using a keyspan adapter (at least on OS X) be aware that it needs a driver installed. Otherwise no ports will be found.

Also note that the attempt to connect to the PR650 must occur within the first few seconds after turning it on.

getLastLum()

This retrieves the luminance (in cd/m^2) from the last call to `.measure()`

getLastSpectrum (*parse=True*)

This retrieves the spectrum from the last call to `.measure()`

If *parse=True* (default): The format is a num array with 100 rows [nm, power]

otherwise: The output will be the raw string from the PR650 and should then be passed to `.parseSpectrumOutput()`. It's more efficient to parse R,G,B strings at once than each individually.

getLum()

Makes a measurement and returns the luminance value

getSpectrum (*parse=True*)

Makes a measurement and returns the current power spectrum

If **parse=True** (default): The format is a num array with 100 rows [nm, power]

If **parse=False** (default): The output will be the raw string from the PR650 and should then be passed to `.parseSpectrumOutput()`. It's slightly more efficient to parse R,G,B strings at once than each individually.

measure (*timeOut=30.0*)

Make a measurement with the device. For a PR650 the device is instructed to make a measurement and then subsequent commands are issued to retrieve info about that measurement.

parseSpectrumOutput (*rawStr*)

Parses the strings from the PR650 as received after sending the command 'd5'. The input argument "rawStr" can be the output from a single phosphor spectrum measurement or a list of 3 such measurements [rawR, rawG, rawB].

sendMessage (*message, timeout=0.5, DEBUG=False*)

Send a command to the photometer and wait an allotted timeout for a response (Timeout should be long for low light measurements)

class `psychopy.hardware.pr.PR655` (*port*)

An interface to the PR655/PR670 via the serial port.

example usage:

```
from psychopy.hardware.pr import PR655
myPR655 = PR655(port)
myPR655.getLum() # make a measurement
nm, power = myPR655.getLastSpectrum() # get a power spectrum for the
last measurement
```

NB `psychopy.hardware.findPhotometer()` will locate and return any supported device for you so you can also do:

```
from psychopy import hardware
phot = hardware.findPhotometer()
print(phot.getLum())
```

Troubleshooting If the device isn't responding try turning it off and turning it on again, and/or disconnecting/reconnecting the USB cable. It may be that the port has become controlled by some other program.

endRemoteMode ()

Puts the colorimeter back into normal mode

getDeviceSN ()

Return the device serial number

getDeviceType ()

Return the device type (e.g. 'PR-655' or 'PR-670')

getLastColorTemp ()

Fetches (from the device) the color temperature (K) of the last measurement

Returns list: status, units, exponent, correlated color temp (Kelvins), CIE 1960 deviation

See also `measure()` automatically populates `pr655.lastColorTemp` with the color temp in Kelvins

getLastSpectrum (*parse=True*)

This retrieves the spectrum from the last call to `measure()`

If *parse=True* (default):

The format is a num array with 100 rows [nm, power]

otherwise:

The output will be the raw string from the PR650 and should then be passed to `parseSpectrumOutput()`. It's more efficient to parse R,G,B strings at once than each individually.

getLastTristim()

Fetches (from the device) the last CIE 1931 Tristimulus values

Returns list: status, units, Tristimulus Values

See also `measure()` automatically populates `pr655.lastTristim` with just the tristimulus coordinates

getLastUV()

Fetches (from the device) the last CIE 1976 u,v coords

Returns list: status, units, Photometric brightness, u, v

See also `measure()` automatically populates `pr655.lastUV` with [u,v]

getLastXY()

Fetches (from the device) the last CIE 1931 x,y coords

Returns list: status, units, Photometric brightness, x,y

See also `measure()` automatically populates `pr655.lastXY` with [x,y]

measure(timeOut=30.0)

Make a measurement with the device.

This automatically populates:

- `.lastLum`
- `.lastSpectrum`
- `.lastCIExy`
- `.lastCIEuv`

parseSpectrumOutput(rawStr)

Parses the strings from the PR650 as received after sending the command 'D5'. The input argument "rawStr" can be the output from a single phosphor spectrum measurement or a list of 3 such measurements [rawR, rawG, rawB].

sendMessage(message, timeout=0.5, DEBUG=False)

Send a command to the photometer and wait an allotted timeout for a response (Timeout should be long for low light measurements)

startRemoteMode()

Sets the Colorimeter into remote mode

8.8.11 pylink (SR research)

For now the SR Research pylink module is packaged with the Standalone flavours of PsychoPy and can be imported with:

```
import pylink
```

You do need to install the Display Software (which they also call Eyelink Developers Kit) for your particular platform. This can be found by following the threads from:

<https://www.sr-support.com/forums/forumdisplay.php?f=17>

for pylink documentation see:

<https://www.sr-support.com/forums/showthread.php?t=14>

`psychopy.hardware.findPhotometer(ports=None, device=None)`

Try to find a connected photometer/photospectrometer!

PsychoPy will sweep a series of serial ports trying to open them. If a port successfully opens then it will try to issue a command to the device. If it responds with one of the expected values then it is assumed to be the appropriate device.

Parameters

ports [a list of ports to search] Each port can be a string (e.g. 'COM1', '/dev/tty.Keyspan1.1') or a number (for win32 comports only). If none are provided then PsychoPy will sweep COM0-10 on win32 and search known likely port names on OS X and linux.

device [string giving expected device (e.g. 'PR650', 'PR655', 'LS110'). If this is not given then an attempt will be made to find a device of any type, but this often fails

Returns

- An object representing the first photometer found
- None if the ports didn't yield a valid response
- None if there were not even any valid ports (suggesting a driver not being installed)

e.g.:

```
# sweeps ports 0 to 10 searching for a PR655
photom = findPhotometer(device='PR655')
print(photom.getLum())
if hasattr(photom, 'getSpectrum'):
    # can retrieve spectrum (e.g. a PR650)
    print(photom.getSpectrum())
```

8.9 psychopy.info - functions for getting information about the system

8.10 psychopy.iohub - ioHub event monitoring framework

ioHub monitors for device events in parallel with the PsychoPy experiment execution by running in a separate process than the main PsychoPy script. This means, for instance, that keyboard and mouse event timing is not quantized by the rate at which the `window.swap()` method is called.

ioHub reports device events to the PsychoPy experiment runtime as they occur. Optionally, events can be saved to a [HDF5](#) file.

All iohub events are timestamped using the PsychoPy global time base (`psychopy.core.getTime()`). Events can be accessed as a device independent event stream, or from a specific device of interest.

A comprehensive set of examples that each use at least one of the iohub devices is available in the `psychopy/demos/coder/iohub` folder.

Note: This documentation is in very early stages of being written. Many sections regarding device usage details are simply placeholders. For information on devices or functionality that has not yet been migrated to the psychopy

documentation, please visit the somewhat outdated [original ioHub doc's](#).

8.10.1 Using psychopy.iohub:

psychopy.iohub Specific Requirements

Computer Specifications

The design / requirements of your experiment itself can obviously influence what the minimum computer specification should be to provide good timing / performance.

The dual process design when running using psychopy.iohub also influences the minimum suggested specifications as follows:

- Intel i5 or i7 CPU. A minimum of **two** CPU cores is needed.
- 8 GB of RAM
- Windows 7 +, OS X 10.7.5 +, or Linux Kernel 2.6 +

Please see the [Recommended hardware](#) section for further information that applies to PsychoPy in general.

Usage Considerations

When using psychopy.iohub, the following constraints should be noted:

1. The pyglet graphics backend must be used; pygame is not supported.
2. ioHub devices that report position data use the unit type defined by the PsychoPy Window. However, position data is reported using the full screen area and size the window was created in. Therefore, for accurate window position reporting, the PsychoPy window must be made full screen.
3. On OS X, Assistive Device support must be enabled when using psychopy.iohub.
 - For OS X 10.7 - 10.8.5, instructions can be found [here](#).
 - For OS X 10.9 +, the program being used to start your experiment script must be specifically authorized. Example instructions on authorizing an OS X 10.9 + app can be viewed [here](#).

Software Requirements

When running PsychoPy using the OS X or Windows standalone distribution, all the necessary python package dependencies have already been installed, so the rest of this section can be skipped.

Note: Hardware specific software may need to be installed depending on the device being used. See the documentation page for the specific device hardware in question for further details.

If psychopy.iohub is being manually installed, first ensure the python packages listed in the [Dependencies](#) section of the manual are installed.

psychopy.iohub requires the following extra dependencies to be installed:

1. [psutil](#) (version 1.2 +) A cross-platform process and system utilities module for Python.
2. [msgpack](#) It's like JSON. but fast and small.

3. [greenlet](#) The greenlet package is a spin-off of Stackless, a version of CPython that supports micro-threads called “tasklets”.
4. [gevent \(version 1.0 or greater\)**](#) A coroutine-based Python networking library.
5. [numexpr](#) Fast numerical array expression evaluator for Python and NumPy.
6. [pytables](#) PyTables is a package for managing hierarchical datasets.
7. [pyYAML](#) PyYAML is a YAML parser and emitter for Python.

Windows installations only

1. [pyHook](#) Python wrapper for global input hooks in Windows.

Linux installations only

1. [python-xlib](#) The Python X11R6 client-side implementation.

OSX installations only

1. [pyobjc](#) : A Python ObjectiveC binding.

Starting the psychopy.iohub Process

To use ioHub within your PsychoPy Coder experiment script, ioHub needs to be started at the start of the experiment script. The easiest way to do this is by calling the `launchHubServer` function.

launchHubServer function

ioHubConnection Class

The `psychopy.iohub.ioHubConnection` object returned from the `launchHubServer` function provides methods for controlling the iohub process and accessing iohub devices and events.

ioHub Devices and Device Events

`psychopy.iohub` supports a large and growing set of supported devices. Details for each device can be found in the following sections.

Keyboard Device

The iohub Keyboard device provides methods to:

- Check for any new keyboard events that have occurred since the last time keyboard events were checked or cleared.
- Wait until a keyboard event occurs.
- Clear the device of any unread events.
- Get a list of all currently pressed keys.

Keyboard Events

The Keyboard device can return two types of events, which represent key press and key release actions on the keyboard.

KeyboardPress Event

KeyboardRelease Event

Mouse Device and Events

TBC

Computer Device

TBC

XInput Gamepad Device and Events

TBC

Eye Tracker Devices and Events

TBC

Serial Port Device and Events

TBC

Analog Input Device and Events

TBC

Touch Screen Device and Events

TBC

8.11 psychopy.logging - control what gets logged

Provides functions for logging error and other messages to one or more files and/or the console, using python's own logging module. Some warning messages and error messages are generated by PsychoPy itself. The user can generate more using the functions in this module.

There are various levels for logged messages with the following order of importance: ERROR, WARNING, DATA, EXP, INFO and DEBUG.

When setting the level for a particular log target (e.g. LogFile) the user can set the minimum level that is required for messages to enter the log. For example, setting a level of INFO will result in INFO, EXP, DATA, WARNING and ERROR messages to be recorded but not DEBUG messages.

By default, PsychoPy will record messages of WARNING level and above to the console. The user can silence that by setting it to receive only CRITICAL messages, (which PsychoPy doesn't use) using the commands:

```
from psychopy import logging
logging.console.setLevel(logging.CRITICAL)
```

class psychopy.logging.**LogFile** (*f=None, level=30, filemode='a', logger=None, encoding='utf8'*)

A text stream to receive inputs from the logging system

Create a log file as a target for logged entries of a given level

Parameters

- **f**: this could be a string to a path, that will be created if it doesn't exist. Alternatively this could be a file object, sys.stdout or any object that supports .write() and .flush() methods
- **level**: The minimum level of importance that a message must have to be logged by this target.
- **mode**: 'a', 'w' Append or overwrite existing log file

setLevel (*level*)

Set a new minimal level for the log file/stream

write (*txt*)

Write directly to the log file (without using logging functions). Useful to send messages that only this file receives

psychopy.logging.**addLevel** (*level, levelName*)

Associate 'levelName' with 'level'.

This is used when converting levels to text during message formatting.

psychopy.logging.**critical** (*message*)

Send the message to any receiver of logging info (e.g. a LogFile) of level *log.CRITICAL* or higher

psychopy.logging.**data** (*msg, t=None, obj=None*)

Log a message about data collection (e.g. a key press)

usage:: log.data(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.DATA* or higher

psychopy.logging.**debug** (*msg, t=None, obj=None*)

Log a debugging message (not likely to be wanted once experiment is finalised)

usage:: log.debug(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.DEBUG* or higher

psychopy.logging.**error** (*message*)

Send the message to any receiver of logging info (e.g. a LogFile) of level *log.ERROR* or higher

psychopy.logging.**exp** (*msg, t=None, obj=None*)

Log a message about the experiment (e.g. a new trial, or end of a stimulus)

usage:: log.exp(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.EXP* or higher

psychopy.logging. (*msg, t=None, obj=None*)

log.critical(message) Send the message to any receiver of logging info (e.g. a LogFile) of level *log.CRITICAL* or higher

psychopy.logging.**flush** (*logger=<psychopy.logging._Logger object>*)

Send current messages in the log to all targets

`psychopy.logging.getLevel (level)`

Return the textual representation of logging level 'level'.

If the level is one of the predefined levels (CRITICAL, ERROR, WARNING, INFO, DEBUG) then you get the corresponding string. If you have associated levels with names using `addLevelName` then the name you have associated with 'level' is returned.

If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

Otherwise, the string "Level %s" % level is returned.

`psychopy.logging.info (msg, t=None, obj=None)`

Log some information - maybe useful, maybe not

usage:: `log.info(message)`

Sends the message to any receiver of logging info (e.g. a `LogFile`) of level `log.INFO` or higher

`psychopy.logging.log (msg, level, t=None, obj=None)`

Log a message

usage:: `log(level, msg, t=t, obj=obj)`

Log the msg, at a given level on the root logger

`psychopy.logging.setDefaultClock (clock)`

Set the default clock to be used to reference all logging times. Must be a `psychopy.core.Clock` object.

Beware that if you reset the clock during the experiment then the resets will be reflected here. That might be useful if you want your logs to be reset on each trial, but probably not.

`psychopy.logging.warn (msg, t=None, obj=None)`

`log.warning(message)`

Sends the message to any receiver of logging info (e.g. a `LogFile`) of level `log.WARNING` or higher

`psychopy.logging.warning (message)`

Sends the message to any receiver of logging info (e.g. a `LogFile`) of level `log.WARNING` or higher

8.11.1 flush()

`psychopy.logging.flush (logger=<psychopy.logging._Logger object>)`

Send current messages in the log to all targets

8.11.2 setDefaultClock()

`psychopy.logging.setDefaultClock (clock)`

Set the default clock to be used to reference all logging times. Must be a `psychopy.core.Clock` object.

Beware that if you reset the clock during the experiment then the resets will be reflected here. That might be useful if you want your logs to be reset on each trial, but probably not.

8.12 psychopy.microphone - Capture and analyze sound

(Available as of version 1.74.00; Advanced features available as of 1.77.00)

8.12.1 Overview

AudioCapture() allows easy audio recording and saving of arbitrary sounds to a file (wav format). AudioCapture will likely be replaced entirely by AdvAudioCapture in the near future.

AdvAudioCapture() can do everything AudioCapture does, and also allows onset-marker sound insertion and detection, loudness computation (RMS audio “power”), and lossless file compression (flac). The Builder microphone component now uses AdvAudioCapture by default.

8.12.2 Audio Capture

8.12.3 Speech recognition

Google’s speech to text API is no longer available. AT&T, IBM, and wit.ai have a similar (paid) service.

8.12.4 Misc

Functions for file-oriented Discrete Fourier Transform and RMS computation are also provided.

8.13 psychopy.misc - miscellaneous routines for converting units etc

Wrapper for all miscellaneous functions and classes from psychopy.tools

psychopy.misc has gradually grown very large and the underlying code for its functions are distributed in multiple files. You can still (at least for now) import the functions here using *from psychopy import misc* but you can also import them from the *tools* sub-modules.

8.13.1 From psychopy.tools.filetools

<i>toFile</i> (filename, data)	Save data (of any sort) as a pickle file.
<i>fromFile</i> (filename)	Load data (of any sort) from a pickle file.
<i>mergeFolder</i> (src, dst[, pattern])	Merge a folder into another.

8.13.2 From psychopy.tools.colorspectools

<i>dkl2rgb</i> (dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.
<i>dklCart2rgb</i> (LUM, LM, S[, conversionMatrix])	Like dkl2rgb except that it uses cartesian coords (LM,S,LUM) rather than sphere
<i>rgb2dklCart</i> (picture[, conversionMatrix])	Convert an RGB image into Cartesian DKL space.
<i>hsv2rgb</i> (hsv_Nx3)	Convert from HSV color space to RGB gun values.
<i>lms2rgb</i> (lms_Nx3[, conversionMatrix])	Convert from cone space (Long, Medium, Short) to RGB.
<i>rgb2lms</i> (rgb_Nx3[, conversionMatrix])	Convert from RGB to cone space (LMS).
<i>dkl2rgb</i> (dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

8.13.3 From psychopy.tools.coordinatestools

<code>cart2pol(x, y[, units])</code>	Convert from cartesian to polar coordinates.
<code>cart2sph(z, y, x)</code>	Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius).
<code>pol2cart(theta, radius[, units])</code>	Convert from polar to cartesian coordinates.
<code>sph2cart(*args)</code>	Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

8.13.4 From `psychopy.tools.monitorunittools`

<code>convertToPix(vertices, pos, units, win)</code>	Takes vertices and position, combines and converts to pixels
<code>cm2pix(cm, monitor)</code>	Convert size in degrees to size in pixels for a given Monitor object
<code>cm2deg(cm, monitor[, correctFlat])</code>	Convert size in cm to size in degrees for a given Monitor object
<code>deg2cm(degrees, monitor[, correctFlat])</code>	Convert size in degrees to size in pixels for a given Monitor object.
<code>deg2pix(degrees, monitor[, correctFlat])</code>	Convert size in degrees to size in pixels for a given Monitor object
<code>pix2cm(pixels, monitor)</code>	Convert size in pixels to size in cm for a given Monitor object
<code>pix2deg(pixels, monitor[, correctFlat])</code>	Convert size in pixels to size in degrees for a given Monitor object

8.13.5 From `psychopy.tools.imagetools`

<code>array2image(a)</code>	Takes an array and returns an image object (PIL)
<code>image2array(im)</code>	Takes an image object (PIL) and returns a numpy array
<code>makeImageAuto(inarray)</code>	Combines float_uint8 and image2array operations ie.

8.13.6 From `psychopy.tools.plottools`

<code>plotFrameIntervals(intervals)</code>	Plot a histogram of the frame intervals.
--	--

8.13.7 From `psychopy.tools.typetools`

<code>float_uint8(inarray)</code>	Converts arrays, lists, tuples and floats ranging -1:1
<code>uint8_float(inarray)</code>	Converts arrays, lists, tuples and UINTs ranging 0:255
<code>float_uint16(inarray)</code>	Converts arrays, lists, tuples and floats ranging -1:1

8.13.8 From `psychopy.tools.unittools`

<code>radians</code>	<code>radians(x[, out])</code>
<code>degrees</code>	<code>degrees(x[, out])</code>

8.14 `psychopy.monitors` - for those that don't like Monitor Center

Most users won't need to use the code here. In general the Monitor Centre interface is sufficient and monitors setup that way can be passed as strings to `Window`s. If there is some aspect of the normal calibration that you wish to override. eg:

```
from psychopy import visual, monitors
mon = monitors.Monitor('SonyG55') #fetch the most recent calib for this monitor
mon.setDistance(114) #further away than normal?
win = visual.Window(size=[1024,768], monitor=mon)
```

You might also want to fetch the `Photometer` class for conducting your own calibrations

8.14.1 Monitor

class psychopy.monitors.**Monitor** (*name*, *width=None*, *distance=None*, *gamma=None*, *notes=None*, *useBits=None*, *verbose=True*, *currentCalib=None*, *autoLog=True*)

Creates a monitor object for storing calibration details. This will be loaded automatically from disk if the monitor name is already defined (see methods).

Many settings from the stored monitor can easily be overridden either by adding them as arguments during the initial call.

arguments:

- *width*, *distance*, *gamma* are details about the calibration
- *notes* is a text field to store any useful info
- *useBits* True, False, None
- *verbose* True, False, None
- **currentCalib** is a dictionary object containing various fields for a calibration. Use with caution since the dictionary may not contain all the necessary fields that a monitor object expects to find.

eg:

```
myMon = Monitor('sony500', distance=114) Fetches the info on the sony500 and overrides its
usual distance to be 114cm for this experiment.
```

These can be saved to the monitor file using `saveMon()` or not (in which case the changes will be lost)

copyCalib (*calibName=None*)

Stores the settings for the current calibration settings as new monitor.

delCalib (*calibName*)

Remove a specific calibration from the current monitor. Won't be finalised unless monitor is saved

gammaIsDefault ()

Determine whether we're using the default gamma values

getCalibDate ()

As a python date object (convert to string using `calibTools.strFromDate`)

getDKL_RGB (*RECOMPUTE=False*)

Returns the DKL->RGB conversion matrix. If one has been saved this will be returned. Otherwise, if power spectra are available for the monitor a matrix will be calculated.

getDistance ()

Returns distance from viewer to the screen in cm, or None if not known

getGamma ()

Returns just the gamma value (not the whole grid)

getGammaGrid ()

Gets the min,max,gamma values for the each gun

getLMS_RGB (*recompute=False*)
Returns the LMS->RGB conversion matrix. If one has been saved this will be returned. Otherwise (if power spectra are available for the monitor) a matrix will be calculated.

getLevelsPost ()
Gets the measured luminance values from last calibration TEST

getLevelsPre ()
Gets the measured luminance values from last calibration

getLinearizeMethod ()
Gets the method that this monitor is using to linearize the guns

getLumsPost ()
Gets the measured luminance values from last calibration TEST

getLumsPre ()
Gets the measured luminance values from last calibration

getMeanLum ()
Returns the mean luminance of the screen if explicitly stored

getNotes ()
Notes about the calibration

getPsychopyVersion ()
Returns the version of PsychoPy that was used to create this calibration

getSizePix ()
Returns the size of the current calibration in pixels, or None if not defined

getSpectra ()
Gets the wavelength values from the last spectrometer measurement (if available)

usage:

- nm, power = monitor.getSpectra()

getUseBits ()
Was this calibration carried out with a bits++ box

getWidth ()
Of the viewable screen in cm, or None if not known

lineariseLums (*desiredLums, newInterpolators=False, overrideGamma=None*)
lums should be uncalibrated luminance values (e.g. a linear ramp) ranging 0:1

newCalib (*calibName=None, width=None, distance=None, gamma=None, notes=None, useBits=False, verbose=True*)
create a new (empty) calibration for this monitor and makes this the current calibration

saveMon ()
Saves the current dictionary of calibrations to disk

setCalibDate (*date=None*)
Sets the current calibration to have a date/time or to the current date/time if none given. (Also returns the date as set)

setCurrent (*calibration=-1*)
Sets the current calibration for this monitor. Note that a single file can hold multiple calibrations each stored under a different key (the date it was taken)

The argument is either a string (naming the calib) or an integer eg:

`myMon.setCurrent('mainCalib')` fetches the calibration named `mainCalib`. You can name calibrations what you want but PsychoPy will give them names of date/time by default. In Monitor Center you can 'copy...' a calibration and give it a new name to keep a second version.

`calibName = myMon.setCurrent(0)` fetches the first calibration (alphabetically) for this monitor

`calibName = myMon.setCurrent(-1)` fetches the last **alphabetical** calibration for this monitor (this is default). If default names are used for calibrations (ie date/time stamp) then this will import the most recent.

setDKL_RGB (*dkl_rgb*)

Sets the DKL->RGB conversion matrix for a chromatically calibrated monitor (matrix is a 3x3 num array).

setDistance (*distance*)

To the screen (cm)

setGamma (*gamma*)

Sets the gamma value(s) for the monitor. This only uses a single gamma value for the three guns, which is fairly approximate. Better to use `setGammaGrid` (which uses one gamma value for each gun)

setGammaGrid (*gammaGrid*)

Sets the min,max,gamma values for the each gun

setLMS_RGB (*lms_rgb*)

Sets the LMS->RGB conversion matrix for a chromatically calibrated monitor (matrix is a 3x3 num array).

setLevelsPost (*levels*)

Sets the last set of luminance values measured AFTER calibration

setLevelsPre (*levels*)

Sets the last set of luminance values measured during calibration

setLineariseMethod (*method*)

Sets the method for linearising 0 uses $y=a+(bx)^{\gamma}$ 1 uses $y=(a+bx)^{\gamma}$ 2 uses linear interpolation over the curve

setLumsPost (*lums*)

Sets the last set of luminance values measured AFTER calibration

setLumsPre (*lums*)

Sets the last set of luminance values measured during calibration

setMeanLum (*meanLum*)

Records the mean luminance (for reference only)

setNotes (*notes*)

For you to store notes about the calibration

setPsychopyVersion (*version*)

To store the version of PsychoPy that this calibration used

setSizePix (*pixels*)

Set the size of the screen in pixels x,y

setSpectra (*nm, rgb*)

Sets the phosphor spectra measured by the spectrometer

setUseBits (*usebits*)

DEPRECATED: Use the new hardware classes to control these devices

setWidth (*width*)

Of the viewable screen (cm)

8.14.2 GammaCalculator

class psychopy.monitors.**GammaCalculator** (*inputs=()*, *lums=()*, *gamma=None*, *bitsIN=8*, *bitsOUT=8*, *eq=1*)

Class for managing gamma tables

Parameters:

- **inputs (required)**= values at which you measured screen luminance either in range 0.0:1.0, or range 0:255. Should include the min and max of the monitor

Then give EITHER “lums” or “gamma”:

- **lums** = measured luminance at given input levels
- **gamma** = your own gamma value (single float)
- **bitsIN** = number of values in your lookup table
- **bitsOUT** = number of bits in the DACs

myTable.gammaModel myTable.gamma

fitGammaErrFun (*params, x, y, minLum, maxLum*)

Provides an error function for fitting gamma function

(used by fitGammaFun)

fitGammaFun (*x, y*)

Fits a gamma function to the monitor calibration data.

Parameters: -xVals are the monitor look-up-table vals, either 0-255 or 0.0-1.0 -yVals are the measured luminances from a photometer/spectrometer

8.14.3 getAllMonitors ()

psychopy.monitors.**getAllMonitors** ()

Find the names of all monitors for which calibration files exist

8.14.4 findPR650 ()

psychopy.monitors.**findPR650** (*ports=None*)

DEPRECATED (as of v.1.60.01). Use *psychopy.hardware.findPhotometer ()* instead, which finds a wider range of devices

8.14.5 getLumSeriesPR650 ()

psychopy.monitors.**getLumSeriesPR650** (*lumLevels=8*, *winSize=(800, 600)*, *monitor=None*, *gamma=1.0*, *allGuns=True*, *useBits=False*, *autoMode='auto'*, *stimSize=0.3*, *photometer='COM1'*)

DEPRECATED (since v1.60.01): Use *psychopy.monitors.getLumSeries ()* instead

8.14.6 `getRGBspectra()`

`psychopy.monitors.getRGBspectra(stimSize=0.3, winSize=(800, 600), photometer='COM1')`

usage: `getRGBspectra(stimSize=0.3, winSize=(800,600), photometer='COM1')`

Params

- 'photometer' could be a photometer object or a serial port name on which a photometer might be found (not recommended)

8.14.7 `gammaFun()`

`psychopy.monitors.gammaFun(xx, minLum, maxLum, gamma, eq=1, a=None, b=None, k=None)`

Returns gamma-transformed luminance values. $y = \text{gammaFun}(x, \text{minLum}, \text{maxLum}, \text{gamma})$

a and b are calculated directly from minLum, maxLum, gamma

Parameters:

- **xx** are the input values (range 0-255 or 0.0-1.0)
- **minLum** = the minimum luminance of your monitor
- **maxLum** = the maximum luminance of your monitor (for this gun)
- **gamma** = the value of gamma (for this gun)

8.14.8 `gammaInvFun()`

`psychopy.monitors.gammaInvFun(yy, minLum, maxLum, gamma, b=None, eq=1)`

Returns inverse gamma function for desired luminance values. $x = \text{gammaInvFun}(y, \text{minLum}, \text{maxLum}, \text{gamma})$

a and b are calculated directly from minLum, maxLum, gamma **Parameters:**

- **xx** are the input values (range 0-255 or 0.0-1.0)
- **minLum** = the minimum luminance of your monitor
- **maxLum** = the maximum luminance of your monitor (for this gun)
- **gamma** = the value of gamma (for this gun)
- **eq determines the gamma equation used;** $\text{eq}=1$ [default]: $yy = a + (b * xx)^{\text{gamma}}$ $\text{eq}=2$: $yy = (a + b * xx)^{\text{gamma}}$

8.14.9 `makeDKL2RGB()`

`psychopy.monitors.makeDKL2RGB(nm, powerRGB)`

Creates a 3x3 DKL->RGB conversion matrix from the spectral input powers

8.14.10 `makeLMS2RGB()`

`psychopy.monitors.makeLMS2RGB(nm, powerRGB)`

Creates a 3x3 LMS->RGB conversion matrix from the spectral input powers

8.15 psychopy.parallel - functions for interacting with the parallel port

This module provides read / write access to the parallel port for Linux or Windows.

The `Parallel` class described below will attempt to load whichever parallel port driver is first found on your system and should suffice in most instances. If you need to use a specific driver then, instead of using `ParallelPort` shown below you can use one of the following as drop-in replacements, forcing the use of a specific driver:

- `psychopy.parallel.PParallelInpOut32`
- `psychopy.parallel.PParallelDLPortIO`
- `psychopy.parallel.PParallelLinux`

Either way, each instance of the class can provide access to a different parallel port.

There is also a legacy API which consists of the routines which are directly in this module. That API assumes you only ever want to use a single parallel port at once.

`psychopy.parallel.ParallelPort`
alias of `PParallelLinux`

8.15.1 Legacy functions

We would strongly recommend you use the class above instead: these are provided for backwards compatibility only.

`parallel.setPortAddress (address=888)`

Set the memory address or device node for your parallel port of your parallel port, to be used in subsequent commands

Common port addresses:

```
LPT1 = 0x0378 or 0x03BC
LPT2 = 0x0278 or 0x0378
LPT3 = 0x0278
```

or for Linux:: `/dev/parport0`

This routine will attempt to find a usable driver depending on your platform

`parallel.setData (data)`

Set the data to be presented on the parallel port (one ubyte). Alternatively you can set the value of each pin (data pins are pins 2-9 inclusive) using `setPin()`

Examples:

```
parallel.setData(0) # sets all pins low
parallel.setData(255) # sets all pins high
parallel.setData(2) # sets just pin 3 high (remember that pin2=bit0)
parallel.setData(3) # sets just pins 2 and 3 high
```

You can also convert base 2 to int v easily in python:

```
parallel.setData(int("00000011", 2)) # pins 2 and 3 high
parallel.setData(int("00000101", 2)) # pins 2 and 4 high
```

`parallel.setPin (pinNumber, state)`

Set a desired pin to be high (1) or low (0).

Only pins 2-9 (incl) are normally used for data output:

```
parallel.setPin(3, 1) # sets pin 3 high
parallel.setPin(3, 0) # sets pin 3 low
```

`parallel.readPin(pinNumber)`

Determine whether a desired (input) pin is high(1) or low(0).

Pins 2-13 and 15 are currently read here

8.16 psychopy.preferences - getting and setting preferences

You can set preferences on a per-experiment basis. For example, if you would like to use a specific [audio library](#), but don't want to touch your user settings in general, you can import preferences and set the option *audioLib* accordingly:

```
from psychopy import prefs
prefs.general['audioLib'] = ['pyo']
from psychopy import sound
```

!!IMPORTANT!! You must import the sound module **AFTER** setting the preferences. To check that you are getting what you want (don't do this in your actual experiment):

```
print sound.Sound
```

The output should be `<class 'psychopy.sound.SoundPyo'>` for pyo, or `<class 'psychopy.sound.SoundPygame'>` for pygame.

You can find the names of the preferences sections and their different options [here](#).

8.16.1 Preferences

Class for loading / saving prefs

class `psychopy.preferences.Preferences`

Users can alter preferences from the dialog box in the application, by editing their user preferences file (which is what the dialog box does) or, within a script, preferences can be controlled like this:

```
import psychopy
psychopy.prefs.general['audioLib'] = ['pyo', 'pygame']
print(prefs)
# prints the location of the user prefs file and all the current vals
```

Use the instance of *prefs*, as above, rather than the *Preferences* class directly if you want to affect the script that's running.

loadAll()

Load the user prefs and the application data

loadUserPrefs()

load user prefs, if any; don't save to a file because doing so will break `easy_install`. Saving to files within the `psychopy/` is fine, eg for key-bindings, but outside it (where user prefs will live) is not allowed by `easy_install` (security risk)

resetPrefs()

removes `userPrefs.cfg`, does not touch `appData.cfg`

restoreBadPrefs(cfg, result)

result = result of validate

saveAppData()

Save the various setting to the appropriate files (or discard, in some cases)

saveUserPrefs()

Validate and save the various setting to the appropriate files (or discard, in some cases)

validate()

Validate (user) preferences and reset invalid settings to defaults

8.17 psychopy.serial - functions for interacting with the serial port

PsychoPy is compatible with Chris Liechti's [pyserial](#) package. You can use it like this:

```
import serial
ser = serial.Serial(0, 19200, timeout=1) # open first serial port
#ser = serial.Serial('/dev/ttyS1', 19200, timeout=1) # or something like this for Mac/Linux machines
ser.write('someCommand')
line = ser.readline() # read a '\n' terminated line
ser.close()
```

Ports are fully configurable with all the options you would expect of RS232 communications. See <http://pyserial.sourceforge.net> for further details and documentation.

pyserial is packaged in the Standalone (Windows and Mac distributions), for manual installations you should install this yourself.

8.18 psychopy.sound - play various forms of sound

8.18.1 Sound

PsychoPy currently supports a choice of two sound libraries: pyo, or pygame. Select which will be used via the [audioLib](#) preference. `sound.Sound()` will then refer to either `SoundPyo` or `SoundPygame`. This can be set on a per-experiment basis by importing preferences, and [setting the audioLib option](#) to use.

It is important to use `sound.Sound()` in order for proper initialization of the relevant sound library. Do not use `sound.SoundPyo` or `sound.SoundPygame` directly. Because they offer slightly different features, the differences between pyo and pygame sounds are described here. Pygame sound is more thoroughly tested, whereas pyo offers lower latency and more features.

8.19 psychopy.tools - miscellaneous tools

Container for all miscellaneous functions and classes

8.19.1 psychopy.tools.colorspectools

Functions and classes related to color space conversion

<code>dkl2rgb(dkl[, conversionMatrix])</code>	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.
<code>dklCart2rgb(LUM, LM, S[, conversionMatrix])</code>	Like dkl2rgb except that it uses cartesian coords (LM,S,LUM) rather than spher

Table 8.23 – continued from previous page

<code>rgb2dklCart</code> (picture[, conversionMatrix])	Convert an RGB image into Cartesian DKL space.
<code>hsv2rgb</code> (hsv_Nx3)	Convert from HSV color space to RGB gun values.
<code>lms2rgb</code> (lms_Nx3[, conversionMatrix])	Convert from cone space (Long, Medium, Short) to RGB.
<code>rgb2lms</code> (rgb_Nx3[, conversionMatrix])	Convert from RGB to cone space (LMS).
<code>dkl2rgb</code> (dkl[, conversionMatrix])	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

Function details

`psychoPy.tools.colorspacetools.dkl2rgb` (dkl, conversionMatrix=None)

Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that this will not be an accurate representation of the color space unless you supply a conversion matrix).

usage:

```
rgb(Nx3) = dkl2rgb(dkl_Nx3(e1,az,radius), conversionMatrix)
rgb(NxNx3) = dkl2rgb(dkl_NxNx3(e1,az,radius), conversionMatrix)
```

`psychoPy.tools.colorspacetools.dklCart2rgb` (LUM, LM, S, conversionMatrix=None)

Like `dkl2rgb` except that it uses cartesian coords (LM,S,LUM) rather than spherical coords for DKL (elev, azim, contr).

NB: this may return rgb values >1 or <-1

`psychoPy.tools.colorspacetools.rgb2dklCart` (picture, conversionMatrix=None)

Convert an RGB image into Cartesian DKL space.

`psychoPy.tools.colorspacetools.hsv2rgb` (hsv_Nx3)

Convert from HSV color space to RGB gun values.

usage:

```
rgb_Nx3 = hsv2rgb(hsv_Nx3)
```

Note that in some uses of HSV space the Hue component is given in radians or cycles (range 0:1]). In this version H is given in degrees (0:360).

Also note that the RGB output ranges -1:1, in keeping with other PsychoPy functions.

`psychoPy.tools.colorspacetools.lms2rgb` (lms_Nx3, conversionMatrix=None)

Convert from cone space (Long, Medium, Short) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that you will not get an accurate representation of the color space unless you supply a conversion matrix)

usage:

```
rgb_Nx3 = lms2rgb(dkl_Nx3(e1,az,radius), conversionMatrix)
```

`psychoPy.tools.colorspacetools.rgb2lms` (rgb_Nx3, conversionMatrix=None)

Convert from RGB to cone space (LMS).

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that you will not get an accurate representation of the color space unless you supply a conversion matrix)

usage:

```
lms_Nx3 = rgb2lms(rgb_Nx3(e1,az,radius), conversionMatrix)
```

`psychopy.tools.colorspectools.dk12rgb(dkl, conversionMatrix=None)`

Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that this will not be an accurate representation of the color space unless you supply a conversion matrix).

usage:

```
rgb(Nx3) = dk12rgb(dkl_Nx3(el,az,radius), conversionMatrix)
rgb(NxNx3) = dk12rgb(dkl_NxNx3(el,az,radius), conversionMatrix)
```

8.19.2 psychopy.tools.coordinatestools

Functions and classes related to coordinate system conversion

<code>cart2pol(x, y[, units])</code>	Convert from cartesian to polar coordinates.
<code>cart2sph(z, y, x)</code>	Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius).
<code>pol2cart(theta, radius[, units])</code>	Convert from polar to cartesian coordinates.
<code>sph2cart(*args)</code>	Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

Function details

`psychopy.tools.coordinatestools.cart2pol(x, y, units='deg')`

Convert from cartesian to polar coordinates.

Usage `theta, radius = pol2cart(x, y, units='deg')`

`units` refers to the units (rad or deg) for `theta` that should be returned

`psychopy.tools.coordinatestools.cart2sph(z, y, x)`

Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius). Output is in degrees.

usage: `array3xN[el,az,rad] = cart2sph(array3xN[x,y,z])` OR `elevation, azimuth, radius = cart2sph(x,y,z)`

If working in DKL space, `z` = Luminance, `y` = S and `x` = LM

`psychopy.tools.coordinatestools.pol2cart(theta, radius, units='deg')`

Convert from polar to cartesian coordinates.

usage:

```
x,y = pol2cart(theta, radius, units='deg')
```

`psychopy.tools.coordinatestools.sph2cart(*args)`

Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

usage: `array3xN[x,y,z] = sph2cart(array3xN[el,az,rad])` OR `x,y,z = sph2cart(elev, azim, radius)`

8.19.3 psychopy.tools.filetools

Functions and classes related to file and directory handling

`psychopy.tools.filetools.toFile(filename, data)`

Save data (of any sort) as a pickle file.

simple wrapper of the cPickle module in core python

`psychopy.tools.filetools.fromFile(filename)`

Load data (of any sort) from a pickle file.

Simple wrapper of the cPickle module in core python

`psychopy.tools.filetools.mergeFolder(src, dst, pattern=None)`

Merge a folder into another.

Existing files in *dst* folder with the same name will be overwritten. Non-existent files/folders will be created.

`psychopy.tools.filetools.openOutputFile(fileName, append=False, delim=None, fileCollisionMethod='rename', encoding='utf-8')`

Open an output file (or standard output) for writing.

Parameters

fileName [string] The desired output file name.

append [bool, optional] If `True`, append data to an existing file; otherwise, overwrite it with new data. Defaults to `True`, i.e. appending.

delim [string, optional] The delimiting character(s) between values. For a CSV file, this would be a comma. For a TSV file, it would be `‘\t’`. Defaults to `‘None’`.

fileCollisionMethod [string, optional] How to handle filename collisions. This is ignored if `append` is set to `True`. Defaults to `rename`.

encoding [string, optional] The encoding to use when writing the file. This parameter will be ignored if `append` is `False` and `fileName` ends with `.psydat` or `.npy` (i.e. if a binary file is to be written). Defaults to `‘utf-8’`.

Returns

f [file] A writable file handle.

Notes

If no known filename extension is given, and the delimiter is a comma, the extension `.csv` will be chosen automatically. If the extension is unknown and the delimiter is a tab, the extension will be `.tsv`. `.txt` will be chosen otherwise.

`psychopy.tools.filetools.genDelimiter(fileName)`

Return a delimiter based on a filename.

Parameters

fileName [string] The output file name.

Returns

delim [string] A delimiter picked based on the supplied filename. This will be `‘,’` if the filename extension is `.csv`, and a tabulator character otherwise.

8.19.4 psychopy.tools.imagetools

Functions and classes related to image handling

<code>array2image(a)</code>	Takes an array and returns an image object (PIL)
-----------------------------	--

Continued on next page

Table 8.25 – continued from previous page

<code>image2array(im)</code>	Takes an image object (PIL) and returns a numpy array
<code>makeImageAuto(inarray)</code>	Combines float_uint8 and image2array operations ie.

Function details

`psychopy.tools.imagetools.array2image(a)`

Takes an array and returns an image object (PIL)

`psychopy.tools.imagetools.image2array(im)`

Takes an image object (PIL) and returns a numpy array

`psychopy.tools.imagetools.makeImageAuto(inarray)`

Combines float_uint8 and image2array operations ie. scales a numeric array from -1:1 to 0:255 and converts to PIL image format

8.19.5 psychopy.tools.monitorunittools

Functions and classes related to unit conversion respective to a particular monitor

<code>convertToPix(vertices, pos, units, win)</code>	Takes vertices and position, combines and converts to pixels
<code>cm2deg(cm, monitor[, correctFlat])</code>	Convert size in cm to size in degrees for a given Monitor object
<code>cm2pix(cm, monitor)</code>	Convert size in degrees to size in pixels for a given Monitor object
<code>deg2cm(degrees, monitor[, correctFlat])</code>	Convert size in degrees to size in pixels for a given Monitor object.
<code>deg2pix(degrees, monitor[, correctFlat])</code>	Convert size in degrees to size in pixels for a given Monitor object
<code>pix2cm(pixels, monitor)</code>	Convert size in pixels to size in cm for a given Monitor object
<code>pix2deg(pixels, monitor[, correctFlat])</code>	Convert size in pixels to size in degrees for a given Monitor object

Function details

`psychopy.tools.monitorunittools.convertToPix(vertices, pos, units, win)`

Takes vertices and position, combines and converts to pixels from any unit

The reason that *pos* and *vertices* are provided separately is that it allows the conversion from deg to apply flat-screen correction to each separately.

The reason that these use function args rather than relying on self.pos is that some stimuli use other terms (e.g. ElementArrayStim uses fieldPos).

`psychopy.tools.monitorunittools.cm2deg(cm, monitor, correctFlat=False)`

Convert size in cm to size in degrees for a given Monitor object

`psychopy.tools.monitorunittools.cm2pix(cm, monitor)`

Convert size in degrees to size in pixels for a given Monitor object

`psychopy.tools.monitorunittools.deg2cm(degrees, monitor, correctFlat=False)`

Convert size in degrees to size in pixels for a given Monitor object.

If *correctFlat* == *False* then the screen will be treated as if all points are equal distance from the eye. This means that each “degree” will be the same size irrespective of its position.

If *correctFlat* == *True* then the *degrees* argument must be an Nx2 matrix for X and Y values (the two cannot be calculated separately in this case).

With *correctFlat* == *True* the positions may look strange because more eccentric vertices will be spaced further apart.

`psychopy.tools.monitorunittools.deg2pix` (*degrees, monitor, correctFlat=False*)

Convert size in degrees to size in pixels for a given Monitor object

`psychopy.tools.monitorunittools.pix2cm` (*pixels, monitor*)

Convert size in pixels to size in cm for a given Monitor object

`psychopy.tools.monitorunittools.pix2deg` (*pixels, monitor, correctFlat=False*)

Convert size in pixels to size in degrees for a given Monitor object

8.19.6 `psychopy.tools.plottools`

Functions and classes related to plotting

`psychopy.tools.plottools.plotFrameIntervals` (*intervals*)

Plot a histogram of the frame intervals.

Where *intervals* is either a filename to a file, saved by `Window.saveFrameIntervals`, or simply a list (or array) of frame intervals

8.19.7 `psychopy.tools.typetools`

Functions and classes related to variable type conversion

`psychopy.tools.typetools.float_uint8` (*inarray*)

Converts arrays, lists, tuples and floats ranging -1:1 into an array of UInt8s ranging 0:255

```
>>> float_uint8(-1)
0
>>> float_uint8(0)
128
```

`psychopy.tools.typetools.uint8_float` (*inarray*)

Converts arrays, lists, tuples and UINTs ranging 0:255 into an array of floats ranging -1:1

```
>>> uint8_float(0)
-1.0
>>> uint8_float(128)
0.0
```

`psychopy.tools.typetools.float_uint16` (*inarray*)

Converts arrays, lists, tuples and floats ranging -1:1 into an array of UInt16s ranging 0:2¹⁶

```
>>> float_uint16(-1)
0
>>> float_uint16(0)
32768
```

8.19.8 `psychopy.tools.unittools`

Functions and classes related to unit conversion

`psychopy.tools.unittools.radians` (*x[, out]*)

Convert angles from degrees to radians.

x [array_like] Input array in degrees.

out [ndarray, optional] Output array of same shape as *x*.

y [ndarray] The corresponding radian values.

deg2rad : equivalent function

Convert a degree array to radians

```
>>> deg = np.arange(12.) * 30.
>>> np.radians(deg)
array([ 0.        ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,
        2.61799388,  3.14159265,  3.66519143,  4.1887902 ,  4.71238898,
        5.23598776,  5.75958653])
```

```
>>> out = np.zeros((deg.shape))
>>> ret = np.radians(deg, out)
>>> ret is out
True
```

psychopy.tools.unittools.**degrees** (*x*[, *out*])

Convert angles from radians to degrees.

x [array_like] Input array in radians.

out [ndarray, optional] Output array of same shape as x.

y [ndarray of floats] The corresponding degree values; if *out* was supplied this is a reference to it.

rad2deg : equivalent function

Convert a radian array to degrees

```
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([ 0.,  30.,  60.,  90., 120., 150., 180., 210., 240.,
       270., 300., 330.])
```

```
>>> out = np.zeros((rad.shape))
>>> r = degrees(rad, out)
>>> np.all(r == out)
True
```

8.20 psychopy.voicekey - Real-time sound processing

(Available as of version 1.83.00)

8.20.1 Overview

Hardware voice-keys are used to detect and signal acoustic properties in real time, e.g., the onset of a spoken word in word-naming studies. PsychoPy provides two virtual voice-keys, one for detecting vocal onsets and one for vocal offsets.

All PsychoPy voice-keys can take their input from a file or from a microphone. Event detection is typically quite similar in both cases.

The base class is very general, and is best thought of as providing a toolkit for developing a wide range of custom voice-keys. It would be possible to develop a set of voice-keys, each optimized for detecting different initial phonemes. Band-pass filtered data and zero-crossing counts are computed in real-time every 2ms.

8.20.2 Voice-Keys

8.20.3 Signal-processing functions

Several utility functions are available for real-time sound analysis.

8.20.4 Sound file I/O

Several helper functions are available for converting and saving sound data from several data formats (numpy arrays, pyo tables) and file formats. All file formats that *pyo* supports are available, including *wav*, *flac* for lossless compression. *mp3* format is not supported (but you can convert to *.wav* using another utility).

8.21 psychopy.web - Web methods

8.21.1 Test for access

`psychopy.web.haveInternetAccess` (*forceCheck=False*)

Detect active internet connection or fail quickly.

If *forceCheck* is *False*, will rely on a cached value if possible.

`psychopy.web.requireInternetAccess` (*forceCheck=False*)

Checks for access to the internet, raise error if no access.

8.21.2 Upload a file over http

`psychopy.web.upload` (*selector, filename, basicAuth=None, host=None, https=False, log=True*)

DEPRECATED: Upload a local file over the internet to a configured http server. Use the requests package instead. See <http://www.python-requests.org/>

This method handshakes with a php script on a remote server to transfer a local file to another machine via http (using POST).

Returns “success” plus a sha256 digest of the file on the server and a byte count. If the upload was not successful, an error code is returned (eg, “too_large” if the file size exceeds the limit specified server-side in *up.php*, or “no_file” if there was no POST attachment).

Note: The server that receives the files needs to be configured before uploading will work. php files and notes for a sys-admin are included in *psychopy/contrib/http/*. In particular, the php script *up.php* needs to be copied to the server’s web-space, with appropriate permissions and directories, including apache basic auth and https (if desired). The maximum size for an upload can be configured within *up.php*

Parameters:

selector [(required, string)] a standard URL of the form *http://host/path/to/up.php*, e.g., *http://upload.psychopy.org/test/up.php*

Note: Limited https support is provided (see below).

filename [(required, string)] the path to the local file to be transferred. The file can be any format: text, utf-8, binary. All files are hex encoded while in transit (increasing the effective file size).

Note: Encryption (*beta*) is available as a separate step. That is, first `encrypt()` the file, then `upload()` the encrypted file in the same way that you would any other file.

basicAuth [(optional)] apache ‘user:password’ string for basic authentication. If a *basicAuth* value is supplied, it will be sent as the auth credentials (in cleartext); using https will encrypt the credentials.

host [(optional)] The default process is to extract host information from the *selector*. The *host* option allows you to specify a host explicitly (i.e., if it differs from the *selector*).

https [(optional)] If the remote server is configured to use https, passing the parameter *https=True* will encrypt the transmission including all data and *basicAuth* credentials. It is approximately as secure as using a self-signed X.509 certificate.

An important caveat is that the authenticity of the certificate returned from the server is not checked, and so the certificate could potentially be spoofed (see the warning under `HTTPSConnection` <http://docs.python.org/library/httplib.html>). Overall, using https can still be much more secure than not using it. The encryption is good, but that of itself does not eliminate all risk. Importantly, it is not as secure as one might expect, given that all major web browsers do check certificate authenticity. The idea behind this parameter is to require people to explicitly indicate that they want to proceed anyway, in effect saying “I know what I am doing and accept the risks (of using un-verified certificates)”.

Author: Jeremy R. Gray, 2012

8.21.3 Proxy set-up and testing

`psychopy.web.setupProxy(log=True)`

Set up the urllib proxy if possible.

The function will use the following methods in order to try and determine proxies:

1. standard `urllib.request.urlopen` (which will use any statically-defined http-proxy settings)
2. previous stored proxy address (in prefs)
3. proxy.pac files if these have been added to system settings
4. auto-detect proxy settings (WPAD technology)

Returns True (success) or False (failure)

Further information:

Troubleshooting

Regrettably, PsychoPy is not bug-free. Running on all possible hardware and all platforms is a big ask. That said, a huge number of bugs have been resolved by the fact that there are literally 1000s of people using the software that have *contributed either bug reports and/or fixes*.

Below are some of the more common problems and their workarounds, as well as advice on how to get further help.

9.1 The application doesn't start

You may find that you try to launch the PsychoPy application, the splash screen appears and then goes away and nothing more happens. What this means is that an error has occurred during startup itself.

Commonly, the problem is that a preferences file is somehow corrupt. To fix that see *Cleaning preferences and app data*, below.

If resetting the preferences files doesn't help then we need to get to an error message in order to work out why the application isn't starting. The way to get that message depends on the platform (see below).

Windows users (starting from the Command Prompt):

1. Did you get an error message that "This application failed to start because the application configuration is incorrect. Reinstalling the application may fix the problem"? If so that indicates you need to [update your .NET installation to SP1](#).
2. **open a DOS Command Prompt (terminal):**
 - (a) go to the Windows Start menu
 - (b) select Run... and type in cmd <Return>
3. paste the following into that window (Ctrl-V doesn't work but you can right-click and select Paste). Replace VERSION with your version number (e.g. 1.61.03):

```
"C:\Program Files\PsychoPy2\python.exe" "C:\Program Files\PsychoPy2\Lib\site-packages\PsychoPy-V
```

4. when you hit <return> you will hopefully get a moderately useful error message that you can *Contribute to the Forum (mailing list)*

Mac users:

1. open the Console app (open spotlight and type console)
2. if there are a huge number of messages there you might find it easiest to clear them (the brush icon) and then start PsychoPy again to generate a new set of messages

9.2 I run a Builder experiment and nothing happens

An error message may have appeared in a dialog box that is hidden (look to see if you have other open windows somewhere).

An error message may have been generated that was sent to output of the Coder view:

1. go to the Coder view (from the Builder>View menu if not visible)
2. if there is no Output panel at the bottom of the window, go to the View menu and select Output
3. try running your experiment again and see if an error message appears in this Output view

If you still don't get an error message but the application still doesn't start then manually turn off the viewing of the Output (as below) and try the above again.

9.3 Manually turn off the viewing of output

Very occasionally an error will occur that crashes the application *after* the application has opened the Coder Output window. In this case the error message is still not sent to the console or command prompt.

To turn off the Output view so that error messages are sent to the command prompt/terminal on startup, open your appData.cfg file (see [Cleaning preferences and app data](#)), find the entry:

```
[coder]
showOutput = True
```

and set it to `showOutput = False` (note the capital 'F').

9.4 Use the source (Luke?)

PsychoPy comes with all the source code included. You may not think you're much of a programmer, but have a go at reading the code. You might find you understand more of it than you think!

To have a look at the source code do one of the following:

- when you get an error message in the [Coder](#) click on the hyperlinked error lines to see the relevant code
- **on Windows**
 - go to *Program Files\PsychoPy2\Lib\site-packages\Psychopy*
 - have a look at some of the files there
- **on Mac**
 - right click the PsychoPy app and select *Show Package Contents*
 - navigate to *Contents/Resources/lib/python2.6/psychopy*

9.5 Cleaning preferences and app data

Every time you shut down PsychoPy (by normal means) your current preferences and the state of the application (the location and state of the windows) are saved to disk. If PsychoPy is crashing during startup you may need to edit those files or delete them completely.

On OS X and Linux the files are:

```
~/psychopy2/appData.cfg  
~/psychopy2/userPrefs.cfg
```

On Windows they are:

```
${DOCS AND SETTINGS}\{USER}\Application Data\psychopy2\appData.cfg  
${DOCS AND SETTINGS}\{USER}\Application Data\psychopy2\userPrefs.cfg
```

The files are simple text, which you should be able to edit in any text editor. Particular changes that you might need to make:

If the problem is that you have a corrupt experiment file or script that is trying and failing to load on startup, you could simply delete the *appData.cfg* file. Please *also Contribute to the Forum (mailing list)* a copy of the file that isn't working so that the underlying cause of the problem can be investigated (google first to see if it's a known issue).

Recipes (“How-to”s)

Below are various tips/tricks/recipes/how-tos for PsychoPy. They involve something that is a little more involved than you would find in FAQs, but too specific for the manual as such (should they be there?).

10.1 Adding external modules to Standalone PsychoPy

You might find that you want to add some additional Python module/package to your Standalone version of PsychoPy. To do this you need to:

- download a copy of the package (make sure it’s for Python 2.7 on your particular platform)
- unzip/open it into a folder
- add that folder to the path of PsychoPy by one of the methods below

Avoid adding the entire path (e.g. the site-packages folder) of separate installation of Python, because that may contain conflicting copies of modules that PsychoPy is also providing.

10.1.1 Using preferences

As of version 1.70.00 you can do this using the PsychoPy preferences/general. There you will find a preference for *paths* which can be set to a list of strings e.g. `['/Users/jwp/code', '~/code/thirdParty']`

These only get added to the Python path when you import psychopy (or one of the psychopy packages) in your script.

10.1.2 Adding a .pth file

An alternative is to add a file into the site-packages folder of your application. This file should be pure text and have the extension .pth to indicate to Python that it adds to the path.

On win32 the site-packages folder will be something like:

`C:/Program Files/PsychoPy2/lib/site-packages`

On OS X you need to right-click the application icon, select ‘Show Package Contents’ and then navigate down to Contents/Resources/lib/python2.6. Put your .pth file here, next to the various libraries.

The advantage of this method is that you don’t need to do the import psychopy step. The downside is that when you update PsychoPy to a new major release you’ll need to repeat this step (patch updates won’t affect it though).

10.2 Animation

General question: How can I animate something?

Conceptually, animation just means that you vary some aspect of the stimulus over time. So the key idea is to draw something slightly different on each frame. This is how movies work, and the same principle can be used to create scrolling text, or fade-in / fade-out effects, and the like.

(copied & pasted from the email list; see the list for people's names and a working script.)

10.3 Scrolling text

Key idea: Vary the **position** of the stimulus across frames.

Question: How can I produce scrolling text (like html's <marquee behavior = "scroll" > directive)?

Answer: PsychoPy has animation capabilities built-in (it can even produce and export movies itself (e.g. if you want to show your stimuli in presentations)). But here you just want to animate stimuli directly.

e.g. create a text stimulus. In the 'pos' (position) field, type:

```
[frameN, 0]
```

and select "set every frame" in the popup button next to that field.

Push the Run button and your text will move from left to right, at one pixel per screen refresh, but stay at a fixed y-coordinate. In essence, you can enter an arbitrary formula in the position field and the stimulus will be-redrawn at a new position on each frame. frameN here refers to the number of frames shown so far, and you can extend the formula to produce what you need.

You might find performance issues (jittering motion) if you try to render a lot of text in one go, in which case you may have to switch to using images of text.

I wanted my text to scroll from right to left. So if you keep your eyes in the middle of the screen the next word to read would come from the right (as if you were actually reading text). The original formula posted above scrolls the other way. So, you have to put a negative sign in front of the formula for it to scroll the other way. You have to change the units to pixel. Also, you have to make sure you have an end time set, otherwise it just flickers. I also set my letter height to 100 pixels. The other problem I had was that I wanted the text to start blank and scroll into the screen. So, I wrote

```
[2000-frameN, 0]
```

and this worked really well.

10.4 Fade-in / fade-out effects

Key idea: vary the **opacity** of the stimulus over frames.

Question: I'd like to present an image with the image appearing progressively and disappearing progressively too. How to do that?

Answer: The Patch stimulus has an opacity field. Set the button next to it to be "set every frame" so that its value can be changed progressively, and enter an equation in the box that does what you want.

e.g. if your screen refresh rate is 60 Hz, then entering:

```
frameN/120
```


would cycle the opacity linearly from 0 to 1.0 over 2s (it will then continue incrementing but it doesn't seem to matter if the value exceeds 1.0).

Using a code component might allow you to do more sophisticated things (e.g. fade in for a while, hold it, then fade out). Or more simply, you just create multiple successive Patch stimulus components, each with a different equation or value in the opacity field depending on their place in the timeline.

10.5 Building an application from your script

A lot of people ask how they can build a standalone application from their Python script. Usually this is because they have a collaborator and want to just send them the experiment.

In general this is not advisable - the resulting bundle of files (single file on OS X) will be on the order of 100Mb and will not provide the end user with any of the options that they might need to control the task (for example, Monitor Center won't be provided so they can't to calibrate their monitor). A better approach in general is to get your collaborator to install the Standalone PsychoPy on their own machine, open your script and press run. (You don't send a copy of Microsoft Word when you send someone a document - you expect the reader to install it themselves and open the document).

Nonetheless, it is technically possible to create exe files on Windows, and Ricky Savjani (savjani at bcm.edu) has kindly provided the following instructions for how to do it. A similar process might be possible on OS X using py2app - if you've done that then feel free to contribute the necessary script or instructions.

10.5.1 Using py2exe to build an executable

Instructions:

1. Download and install py2exe (<http://www.py2exe.org/>)
2. Develop your PsychoPy script as normal
3. Copy this setup.py file into the same directory as your script
4. Change the Name of progName variable in this file to the Name of your desired executable program name
5. Use cmd (or bash, terminal, etc.) and run the following in the directory of your the two files: python setup.py py2exe
6. Open the 'dist' directory and run your executable

A example setup.py script:

```
# Created 8-09-2011
# Ricky Savjani
# (savjani at bcm.edu)

#import necessary packages
from distutils.core import setup
import os, matplotlib
import py2exe

#the name of your .exe file
progName = 'MultipleSchizophrenia.py'

#Initialize Holder Files
preference_files = []
app_files = []
my_data_files=matplotlib.get_py2exe_datafiles()
```

```
#define which files you want to copy for data_files
for files in os.listdir('C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.00-py2.6.egg\\psychopy\\pr
    fl = 'C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.00-py2.6.egg\\psychopy\\pr
    preference_files.append(fl)

#if you might need to import the app files
#for files in os.listdir('C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.00-py2.6.egg\\psychopy\\ap
#    fl = 'C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.00-py2.6.egg\\psychopy\\ap
#    app_files.append(fl)

all_files = [("psychopy\\preferences", preference_files), ("psychopy\\app", app_files), my_data_files

#combine the files
all_files = [("psychopy\\preferences", preference_files), my_data_files[0]]

#define the setup
setup(
    console=[progName],
    data_files = all_files,
    options = {
        "py2exe":{
            "skip_archive": True,
            "optimize": 2
        }
    }
)
```

10.6 Builder - providing feedback

If you're using the Builder then the way to provide feedback is with a *Code Component* to generate an appropriate message (and then a *Text Component* to present that message). PsychoPy will be keeping track of various aspects of the stimuli and responses for you throughout the experiment and the key is knowing where to find those.

The following examples assume you have a *Loop* called *trials*, containing a *Routine* with a *Keyboard Component* called *key_resp*. Obviously these need to be adapted in the code below to fit your experiment.

Note: The following generate strings use python 'formatted strings'. These are very powerful and flexible but a little strange when you aren't used to them (they contain odd characters like `%2f`). See *Generating formatted strings* for more info.

10.6.1 Feedback after a trial

This is actually demonstrated in the demo, *ExtendedStroop* (in the Builder>demos menu, unpack the demos and then look in the menu again. tada!)

If you have a Keyboard Component called *key_resp* then, after every trial you will have the following variables:

```
key_resp.keys #a python list of keys pressed
key_resp.rt #the time to the first key press
key_resp.corr #None, 0 or 1, if you are using 'store correct'
```

To create your *msg*, insert the following into the 'start experiment' section of the *Code Component*:

```
msg='doh!'#if this comes up we forgot to update the msg!
```

and then insert the following into the *Begin Routine* section (this will get run every repeat of the routine):

```
if len(key_resp.keys)<1:
    msg="Failed to respond"
elif resp.corr:#stored on last run routine
    msg="Correct! RT=%3f" %(resp.rt)
else:
    msg="Oops! That was wrong"
```

10.6.2 Feedback after a block

In this case the feedback routine would need to come after the loop (the block of trials) and the message needs to use the stored data from the loop rather than the *key_resp* directly. Accessing the data from a loop is not well documented but totally possible.

In this case, to get all the keys pressed in a *numpy* array:

```
trials.data['key_resp.keys'] #numpy array with size=[ntrials,ntypes]
```

If you used the ‘Store Correct’ feature of the Keyboard Component (and told psychopy what the correct answer was) you will also have a variable:

```
#numpy array storing whether each response was correct (1) or not (0)
trials.data['resp.corr']
```

So, to create your *msg*, insert the following into the ‘start experiment’ section of the *Code Component*:

```
msg='doh!'#if this comes up we forgot to update the msg!
```

and then insert the following into the *Begin Routine* section (this will get run every repeat of the routine):

```
nCorr = trials.data['key_resp.corr'].sum() #.std(), .mean() also available
meanRt = trials.data['key_resp.rt'].mean()
msg = "You got %i trials correct (rt=%2f)" %(nCorr,meanRt)
```

10.6.3 Draw your message to the screen

Using one of the above methods to generate your *msg* in a *Code Component*, you then need to present it to the participant by adding a *Text Component* to your *feedback* Routine and setting its text to *\$msg*.

Warning: The Text Component needs to be below the Code Component in the Routine (because it needs to be updated after the code has been run) and it needs to *set every repeat*.

10.7 Builder - terminating a loop

People often want to terminate their *Loops* before they reach the designated number of trials based on subjects’ responses. For example, you might want to use a Loop to repeat a sequence of images that you want to continue until a key is pressed, or use it to continue a training period, until a criterion performance is reached.

To do this you need a *Code Component* inserted into your *routine*. All loops have an attribute called *finished* which is set to *True* or *False* (in Python these are really just other names for *1* and *0*). This *finished* property gets checked on each pass through the loop. So the key piece of code to end a loop called *trials* is simply:

```
trials.finished=True #or trials.finished=1 if you prefer
```

Of course you need to check the condition for that with some form of *if* statement.

Example 1: You have a change-blindness study in which a pair of images flashes on and off, with intervening blanks, in a loop called *presentationLoop*. You record the key press of the subject with a *Keyboard Component* called *resp1*. Using the ‘ForceEndTrial’ parameter of *resp1* you can end the current cycle of the loop but to end the loop itself you would need a *Code Component*. Insert the following two lines in the *End Routine* parameter for the Code Component, which will test whether more than zero keys have been pressed:

```
if len(resp1.keys)>0:
    presentationLoop.finished=1
```

Example 2: Sometimes you may have more possible trials than you can actually display. By default, a loop will present all possible trials (*nReps* * *length-of-list*). If you only want to present the first 10 of all possible trials, you can use a code component to count how many have been shown, and then finish the loop after doing 10.

This example assumes that your loop is named ‘trials’. You need to add two things, the first to initialize the count, and the second to update and check it.

Begin Experiment:

```
myCount = 0
```

Begin Routine:

```
myCount = myCount + 1
if myCount > 10:
    trials.finished = True
```

Note: In Python there is no *end* to finish an *if* statement. The content of the *if* or of a for-loop is determined by the indentation of the lines. In the above example only one line was indented so that one line will be executed if the statement evaluates to *True*.

10.8 Installing PsychoPy in a classroom (administrators)

For running PsychoPy in a classroom environment it is probably preferable to have a ‘partial’ network installation. The PsychoPy library features frequent new releases, including bug fixes and you want to be able to update machines with these new releases. But PsychoPy depends on many other python libraries (over 200Mb in total) that tend not to change so rapidly, or at least not in ways critical to the running of experiments. If you install the whole PsychoPy application on the network then all of this data has to pass backwards and forwards, and starting the app will take even longer than normal.

The basic aim of this document is to get to a state whereby;

- Python and the major dependencies of PsychoPy are installed on the local machine (probably a disk image to be copied across your lab computers)
- PsychoPy itself (only ~2Mb) is installed in a network location where it can be updated easily by the administrator
- a file is created in the installation that provides the path to the network drive location
- Start-Menu shortcuts need to be set to point to the local Python but the remote PsychoPy application launcher

Once this is done, the vast majority of updates can be performed simply by replacing the PsychoPy library on the network drive.

10.8.1 1. Install dependencies locally

Download the latest version of the Standalone PsychoPy distribution, and run as administrator. This will install a copy of Python and many dependencies to a default location of

C:\Program Files\PsychoPy2

10.8.2 2. Move the PsychoPy to the network

You need a network location that is going to be available, with read-only access, to all users on your machines. You will find all the contents of PsychoPy itself at something like this (version dependent obviously):

C:\Program Files\PsychoPy2\Lib\site-packages\PsychoPy-1.70.00-py2.6.egg

Move that entire folder to your network location and call it psychopyLib (or similar, getting rid of the version-specific part of the name). Now the following should be a valid path:

<NETWORK_LOC>\psychopyLib\psychopy

10.8.3 3. Update the Python path

The Python installation (in C:\Program Files\PsychoPy2) needs to know about the network location. If Python finds a text file with extension *.pth* anywhere on its existing path then it will add to the path any valid paths it finds in the file. So create a text file that has one line in it:

<NETWORK_LOC>\psychopyLib

You can test if this has worked. Go to *C:\Program Files\PsychoPy2* and double-click on python.exe. You should get a Python terminal window come up. Now try:

```
>>> import psychopy
```

If psychopy is not found on the path then there will be an import error. Try adjusting the *.pth* file, restarting python.exe and importing again.

10.8.4 4. Update the Start Menu

The shortcut in the Windows Start Menu will still be pointing to the local (now non-existent) PsychoPy library. Right-click it to change properties and set the shortcut to point to something like:

```
"C:\Program Files\PsychoPy2\pythonw.exe" "<NETWORK_LOC>\psychopyLib\psychopy\app\psychopyApp.py"
```

You probably spotted from this that the PsychoPy app is simply a Python script. You may want to update the file associations too, so that *.psyexp* and *.py* are opened with:

```
"C:\Program Files\PsychoPy2\pythonw.exe" "<NETWORK_LOC>\psychopyLib\psychopy\app\psychopyApp.py" "%1"
```

Lastly, to make the shortcut look pretty, you might want to update the icon too. Set the icon's location to:

```
"<NETWORK_LOC>\psychopyLib\psychopy\app\Resources\psychopy.ico"
```

10.8.5 5. Updating to a new version

Fetch the latest .zip release. Unpack it and replace the contents of *<NETWORK_LOC>\psychopyLib* with the contents of the zip file.

10.9 Generating formatted strings

A formatted string is a variable which has been converted into a string (text). In python the specifics of how this is done is determined by what kind of variable you want to print.

Example 1: You have an experiment which generates a string variable called *text*. You want to insert this variable into a string so you can print it. This would be achieved with the following code:

```
message = 'The result is %s' %(text)
```

This will produce a variable *message* which if used in a text object would print the phrase ‘The result is’ followed by the variable *text*. In this instance %s is used as the variable being entered is a string. This is a marker which tells the script where the variable should be entered. %text tells the script which variable should be entered there.

Multiple formatted strings (of potentially different types) can be entered into one string object:

```
longMessage = 'Well done %s that took %0.3f seconds' %(info['name'], time)
```

Some of the handy formatted string types:

```
>>> x=5
>>> x1=5124
>>> z='someText'
>>> 'show %s' %(z)
'show someText'
>>> '%0.1f' %(x)    #will show as a float to one decimal place
'5.0'
>>> '%3i' %(x)      #an integer, at least 3 chars wide, padded with spaces
'  5'
>>> '%03i' %(x)     #as above but pad with zeros (good for participant numbers)
'005'
```

See the [python documentation](#) for a more complete list.

10.10 Coder - interleave staircases

Often psychophysicists using staircase procedures want to interleave multiple staircases, either with different start points, or for different conditions.

There is now a class, *psychopy.data.MultiStairHandler* to allow simple access to interleaved staircases of either basic or QUEST types. That can also be used from the *Loops* in the *Builder*. The following method allows the same to be created in your own code, for greater options.

The method works by nesting a pair of loops, one to loop through the number of trials and another to loop across the staircases. The staircases can be shuffled between trials, so that they do not simply alternate.

Note: Note the need to create a *copy* of the info. If you simply do *thisInfo=info* then all your staircases will end up pointing to the same object, and when you change the info in the final one, you will be changing it for all.

```
from psychopy import visual, core, data, event
from numpy.random import shuffle
import copy, time #from the std python libs

#create some info to store with the data
info={}
info['startPoints']=[1.5,3,6]
```

```

info['nTrials']=10
info['observer']='jwp'

win=visual.Window([400,400])
#-----
#create the stimuli
#-----

#create staircases
stairs=[]
for thisStart in info['startPoints']:
    #we need a COPY of the info for each staircase
    #(or the changes here will be made to all the other staircases)
    thisInfo = copy.copy(info)
    #now add any specific info for this staircase
    thisInfo['thisStart']=thisStart #we might want to keep track of this
    thisStair = data.StairHandler(startVal=thisStart,
        extraInfo=thisInfo,
        nTrials=50, nUp=1, nDown=3,
        minVal = 0.5, maxVal=8,
        stepSizes=[4,4,2,2,1,1])
    stairs.append(thisStair)

for trialN in range(info['nTrials']):
    shuffle(stairs) #this shuffles 'in place' (ie stairs itself is changed, nothing returned)
    #then loop through our randomised order of staircases for this repeat
    for thisStair in stairs:
        thisIntensity = thisStair.next()
        print 'start=%.2f, current=%.4f' %(thisStair.extraInfo['thisStart'], thisIntensity)

        #-----
        #run your trial and get an input
        #-----
        keys = event.waitKeys() #(we can simulate by pushing left for 'correct')
        if 'left' in keys: wasCorrect=True
        else: wasCorrect = False

        thisStair.addData(wasCorrect) #so that the staircase adjusts itself

    #this trial (of all staircases) has finished
#all trials finished

#save data (separate pickle and txt files for each staircase)
dateStr = time.strftime("%b_%d_%H%M", time.localtime())#add the current time
for thisStair in stairs:
    #create a filename based on the subject and start value
    filename = "%s start%.2f %s" %(thisStair.extraInfo['observer'], thisStair.extraInfo['thisStart'],
    thisStair.saveAsPickle(filename)
    thisStair.saveAsText(filename)

```

10.11 Making isoluminant stimuli

From the mailing list (see there for names, etc):

Q1: How can I create colours (RGB) that are isoluminant?

A1: The easiest way to create isoluminant stimuli (or control the luminance content) is to create the stimuli in DKL

space and then convert them into RGB space for presentation on the monitor.

More details on DKL space can be found in the section about [Color spaces](#) and conversions between DKL and RGB can be found in the API reference for [psychopy.misc](#)

Q2: There's a difference in luminance between my stimuli. How could I correct for that?

I'm running an experiment where I manipulate color chromatic saturation, keeping luminance constant. I've coded the colors (red and blue) in rgb255 for 6 saturation values (10%, 20%, 30%, 40%, 50%, 60%, 90%) using a conversion from HSL to RGB color space.

Note that we don't possess spectrophotometers such as PR650 in our lab to calibrate each color gun. I've calibrated the gamma of my monitor psychophysically. Gamma was set to 1.7 (threshold) for `gamma(lum)`, `gamma(R)`, `gamma(G)`, `gamma(B)`. Then I've measured the luminance of each stimuli with a Brontes colorimeter. But there's a difference in luminance between my stimuli. How could I correct for that?

A2: Without a spectroradiometer you won't be able to use the color spaces like DKL which are designed to help this sort of thing.

If you don't care about using a specific colour space though you should be able to deduce a series of isoluminant colors manually, because the luminance outputs from each gun should sum linearly. e.g. on my monitor:

```
maxR=46cd/m2
maxG=114
maxB=15
```

(note that green is nearly always brightest)

So I could make a 15cd/m2 stimulus using various appropriate fractions of those max values (requires that the screen is genuinely gamma-corrected):

```
R=0, G=0, B=255
R=255*15/46, G=0, B=0
R=255*7.5/46, G=255*15/114, B=0
```

Note that, if you want a pure fully-saturated blue, then you're limited by the monitor to how bright you can make your stimulus. If you want brighter colours your blue will need to include some of the other guns (similarly for green if you want to go above the max luminance for that gun).

A2.1. You should also consider that even if you set appropriate RGB values to display your pairs of chromatic stimuli at the same luminance that they might still appear different, particularly between observers (and even if your light measurement device says the luminance is the same, and regardless of the colour space you want to work in). To make the pairs perceptually isoluminant, each observer should really determine their own isoluminant point. You can do this with the minimum motion technique or with heterochromatic flicker photometry.

10.12 Adding a web-cam

From the mailing list (see there for names, etc):

"I spent some time today trying to get a webcam feed into my psychopy proj, inside my visual.window. The solution involved using the opencv module, capturing the image, converting that to PIL, and then feeding the PIL into a SimpleImageStim and looping and win.flipping. Also, to avoid looking like an Avatar in my case, you will have to change the default decoder used in PIL fromstring to utilize BGR instead of RGB in the decoding. I thought I would save some time for people in the future who might be interested in using a webcam feed for their psychopy project. All you need to do is import the opencv module into psychopy (importing modules was well documented by psychopy online) and integrate something like this into your psychopy script."


```

from psychopy import visual, event, core
import Image, time, pylab, cv, numpy

mywin = visual.Window(allowGUI=False, monitor='testMonitor', units='norm', colorSpace='rgb', color=[-1,
mywin.setMouseVisible(False)

capture = cv.CaptureFromCAM(0)
img = cv.QueryFrame(capture)
pi = Image.fromstring("RGB", cv.GetSize(img), img.tostring(), "raw", "BGR", 0, 1)
print pi.size
myStim = visual.GratingStim(win=mywin, tex=pi, pos=[0,0.5], size = [0.6,0.6], opacity = 1.0, units =
myStim.setAutoDraw(True)

while True:
    img = cv.QueryFrame(capture)
    pi = Image.fromstring("RGB", cv.GetSize(img), img.tostring(), "raw", "BGR", 0, 1)
    myStim.setTex(pi)
    mywin.flip()
    theKey = event.getKeys()
    if len(theKey) != 0:
        break

```

Frequently Asked Questions (FAQs)

11.1 Why is the bits++ demo not working?

So far PsychoPy supports bits++ only in the bits++ mode (rather than mono++ or color++). In this mode, a code (the T-lock code) is written to the lookup table on the bits++ device by drawing a line at the top of the window. The most likely reason that the demo isn't working for you is that this line is not being detected by the device, and so the lookup table is not being modified. Most of these problems are actually nothing to do with PsychoPy /per se/, but to do with your graphics card and the CRS bits++ box itself.

There are a number of reasons why the T-lock code is not being recognised:

- the bits++ device is in the wrong mode. Open the utility that CRS supply and make sure you're in the right mode. Try resetting the bits++ (turn it off and on).
- the T-lock code is not fully on the screen. If you create a window that's too big for the screen or badly positioned then the code will be broken/not visible to the device.
- the T-lock code is on an 'odd' pixel.
- the graphics card is doing some additional filtering (win32). Make sure you turn off any filtering in the advanced display properties for your graphics card
- the gamma table of the graphics card is not set to be linear (but this should normally be handled by PsychoPy, so don't worry so much about it).
- you've got a Mac that's performing temporal dithering (new Macs, around 2009). Apple have come up with a new, very annoying idea, where they continuously vary the pixel values coming out of the graphics card every frame to create additional intermediate colours. This will break the T-lock code on 1/2-2/3rds of frames.

11.2 Can PsychoPy run my experiment with sub-millisecond timing?

This question is common enough and complex enough to have a section of the manual all of its own. See *Timing Issues and synchronisation*

Resources (e.g. for teaching)

There are a number of further resources to help learn/teach about PsychoPy.

If you also have PsychoPy materials/course then please let us know so that we can link to them from here too!

12.1 Workshops

- **P4N: Python for Neuroscience** There is usually a 3-day Python bootcamp at Nottingham University in April. It won't be *only* about PsychoPy, but about Python for science more generally and focussing on coding rather than using the Builder interface. For further info see P4N

12.2 Youtube tutorials

- [Youtube PsychoPy tutorial](#) showing how to build a basic experiment in the *Builder* interface. That's a great way to get started; build your own complete experiment in 15 minutes flat!
- There's also a [subtitled version of the stroop video tutorial](#) (Thanks Kevin Cole for doing that!)

12.3 Materials for Builder

- At [School of Psychology, University of Nottingham](#), PsychoPy is now used for all first year practical class teaching. The classes that comprise that first year course are provided below. They were created partially with funding from the former [Higher Education Academy Psychology Network](#). Note that the materials here will be updated frequently as they are further developed (e.g. to update screenshots etc) so make sure you have the latest version of them!

PsychoPy_pracs_2011v2.zip (21MB) (last updated: 15 Dec 2011)

- The [GestaltReVision group \(University of Leuven\)](#) wiki covering PsychoPy (some Builder info and great tutorials for Python/PsychoPy coding of experiments).

12.4 Materials for Coder

- [Gary Lupyan](#) runs a class on programming experiments using Python/PsychoPy and makes his lecture materials available [on this wiki](#)

- The [GestaltReVision](#) group (University of Leuven) offers a three-day crash course to Python and PsychoPy on a [IPython Notebook](#), and has lots of great information taking you from basic programming to advanced techniques.
 - Radboud University, Nijmegen also has a [PsychoPy programming course](#)
 - [Programming for Psychology in Python - Vision Science](#) has lessons and screencasts on PsychoPy (Damien Mannion, UNSW Australia).
-

12.5 Previous events

- ECEM, August 2013 : Python for eye-tracking workshop with (Sol Simpson, Michael MacAskill and Jon Peirce). Download [Python-for-eye-tracking materials](#)
- VSS
- [Yale, 21-23 July](#) : The first ever dedicated PsychoPy workshop/conference was at Yale, 21-23 July 2011. Thanks Jeremy for organising!
- EPS Satellite workshop, 8 July 2011
- [BPS Maths Stats and Computing Section](#) workshop (Dec 2010):

For developers:

For Developers

There is a separate mailing list to discuss development ideas and issues.

For developers the best way to use PsychoPy is to install a version to your own copy of python (preferably 2.6 but 2.5 is OK). Make sure you have all the *Dependencies*, including the extra *Suggested packages* for developers.

Don't *install* PsychoPy. Instead fetch a copy of the git repository and add this to the python path using a .pth file. Other users of the computer might have their own standalone versions installed without your repository version touching them.

13.1 Using the repository

Note: Much of the following is explained with more detail in the [nitime documentation](#), and then in further detail in numerous online tutorials.

13.1.1 Workflow

The use of git and the following workflow allows people to contribute changes that can easily be incorporated back into the project, while (hopefully) maintaining order and consistency in the code. All changes should be tracked and reversible.

- Create a fork of the central psychopy/psychopy repository
- Create a local clone of that fork
- **For small changes**
 - make the changes directly in the master branch
 - push back to your fork
 - submit a pull request to the central repository
- **For substantial changes (new features)**
 - create a branch
 - when finished run unit tests
 - when the unit tests pass merge changes back into the *master* branch
 - submit a pull request to the central repository

13.1.2 Create your own fork of the central repository

Go to [github](#), create an account and make a fork of the [psychopy repository](#). You can change your fork in any way you choose without it affecting the central project. You can also share your fork with others, including the central project.

13.1.3 Fetch a local copy

Install [git on your computer](#). Create and upload an ssh key to your github account - this is necessary for you to push changes back to your fork of the project at github.

Then, in a folder of your choosing fetch your fork:

```
$ git clone git@github.com:USER/psychopy.git
$ cd psychopy
$ git remote add upstream git://github.com/psychopy/psychopy.git
```

The last line connects your copy (with read access) to the central server so you can easily fetch any updates to the central repository.

13.1.4 Fetching the latest version

Periodically it's worth fetching any changes to the central psychopy repository (into your *master* branch, more on that below):

```
$ git checkout master
$ git pull upstream master # here 'master' is the desired branch of psychopy to fetch
```

13.1.5 Run PsychoPy using your local copy

Now that you've fetched the latest version of psychopy using git, you should run this version in order to try out yours/others latest improvements. See [this guide](#) on how to permanently run your git version of psychopy instead of the version you previously installed.

Run git version for just one session (Linux and Mac only): If you want to switch between the latest-and-greatest development version from git and the stable version installed on your system, you can choose to only temporarily run the git version. Open a terminal and set a temporary python path to your psychopy git folder:

```
$ export PYTHONPATH=/path/to/local/git/folder/
```

To check that worked you should open python in the terminal and try to import psychopy:

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import psychopy
```

PsychoPy depends on a lot of other packages and you may get a variety of failures to import them until you have them all installed in your custom environment!

13.1.6 Fixing bugs and making minor improvements

You can make minor changes directly in the *master* branch of your fork. After making a change you need to *commit* a set of changes to your files with a message. This enables you to group together changes and you will subsequently be able to go back to any previous *commit*, so your changes are reversible.

I (Jon) usually do this by opening the graphical user interface that comes with git:

```
$ git gui
```

From the GUI you can select (or *stage* in git terminology) the files that you want to include in this particular *commit* and give it a message. Give a clear summary of the changes for the first line. You can add more details about the changes on lower lines if needed.

If you have internet access then you could also push your changes back up to your fork (which is called your *origin* by default), either by pressing the *push* button in the GUI or by closing that and typing:

```
$ git push
```

13.1.7 Commit messages

Informative commit messages are really useful when we have to go back through the repository finding the time that a particular change to the code occurred. Precede your message with one or more of the following to help us spot easily if this is a bug fix (which might need pulling into other development branches) or new feature (which we might want to avoid pulling in if it might disrupt existing code).

- *BF* : bug fix
- *FF* : ‘feature’ fix. This is for fixes to code that hasn’t been released
- *RF* : refactoring
- *NF* : new feature
- *ENH* : enhancement (improvement to existing code)
- *DOC*: for all kinds of documentation related commits
- *TEST*: for adding or changing tests

NB: The difference between BF and FF is that BF indicates a fix that is appropriate for back-porting to earlier versions, whereas FF indicates a fix to code that has not been released, and so cannot be back-ported.

13.1.8 Share your improvement with others

Only a couple of people have direct write-access to the psychopy repository, but you can get your changes included in *upstream* by pushing your changes back to your github fork and then [submitting a pull request](#). Communication is good, and hopefully you have already been in touch (via the user or dev lists) about your changes.

When adding an improvement or new feature, consider how it might impact others. Is it likely to be generally useful, or is it something that only you or your lab would need? (It’s fun to contribute, but consider: does it actually need to be part of PsychoPy?) Including more features has a downside in terms of complexity and bloat, so try to be sure that there is a “business case” for including it. If there is, try at all times to be backwards compatible, e.g., by adding a new keyword argument to a method or function (not always possible). If it’s not possible, it’s crucial to get wider input about the possible impacts. Flag situations that would break existing user scripts in your commit messages.

Part of sharing your code means making things sensible to others, which includes good coding style and writing some documentation. You are the expert on your feature, and so are in the best position to elaborate nuances or gotchas. Use meaningful variable names, and include comments in the code to explain non-trivial things, especially the intention

behind specific choices. Include or edit the appropriate doc-string, because these are automatically turned into API documentation (via sphinx). Include doc-tests if that would be meaningful. The existing code base has a comment / code ratio of about 28%, which earns it high marks.

For larger changes and especially new features, you might need to create some usage examples, such as a new Coder demo, or even a Builder demo. These can be invaluable for being a starting point from which people can adapt things to the needs of their own situation. This is a good place to elaborate usage-related gotchas.

In terms of style, try to make your code blend in with and look like the existing code (e.g., using about the same level of comments, use camelCase for var names, despite the conflict with the usual PEP – we’ll eventually move to the underscore style, but for now keep everything consistent within the code base). In your own code, write however you like of course. This is just about when contributing to the project.

13.1.9 Add a new feature branch

For more substantial work, you should create a new branch in your repository. Often while working on a new feature other aspects of the code will get broken and the *master* branch should always be in a working state. To create a new branch:

```
$ git branch feature-somethingNew
```

You can now switch to your new feature branch with:

```
$ git checkout feature-somethingNew
```

And get back to your *master* branch with:

```
$ git checkout master
```

You can push your new branch back to your fork (*origin*) with:

```
$ git push origin feature-somethingNew
```

13.1.10 Completing work on a feature

When you’re done run the unit tests for your feature branch. Set the *debug* preference setting (in the app section) to True, and restart psychopy. This will enable access to the test-suite. In debug mode, from the Coder (not Builder) you can now do Ctrl-T / Cmd-T (see Tools menu, Unit Testing) to bring up the unit test window. You can select a subset of tests to run, or run them all.

It’s also possible to run just selected tests, such as doctests within a single file. From a terminal window:

```
cd psychopy/tests/ #eg /Users/jgray/code/psychopy/psychopy/tests
./run.py path/to/file_with_doctests.py
```

If the tests pass you hopefully haven’t damaged other parts of PsychoPy (!?). If possible add a unit test for your new feature too, so that if other people make changes they don’t break your work!

You can merge your changes back into your master branch with:

```
$ git checkout master
$ git merge feature-somethingNew
```

Merge conflicts happen, and need to be resolved. If you configure your git preferences (*~/.gitconfig*) to include:

```
[merge]
    summary = true
    log = true
    tool = opendiff
```

then you'll be able to use a handy GUI interface (opendiff) for reviewing differences and conflicts, just by typing:

```
git mergetool
```

from the command line after hitting a merge conflict (such as during a *git pull upstream master*).

Once you've folded your new code back into your master and pushed it back to your github fork then it's time to *Share your improvement with others*.

13.2 Adding documentation

There are several ways to add documentation, all of them useful: doc strings, comments in the code, and demos to show an example of actual usage. To further explain something to end-users, you can create or edit a .rst file that will automatically become formatted for the web, and eventually appear on www.psychopy.org.

You make a new file under `psychopy/docs/source/`, either as a new file or folder or within an existing one.

To test that your doc source code (.rst file) does what you expect in terms of formatting for display on the web, you can simply do something like (this is my actual path, unlikely to be yours):

```
$ cd /Users/jgray/code/psychopy/docs/
$ make html
```

Do this within your docs directory (requires sphinx to be installed, try "easy_install sphinx" if it's not working). That will add a `build/html` sub-directory.

Then you can view your new doc in a browser, e.g., for me:

```
file:///Users/jgray/code/psychopy/docs/build/html/
```

Push your changes to your github repository (using a "DOC:" commit message) and let Jon know, e.g. with a pull request.

13.3 Adding a new Builder Component

Builder Components are auto-detected and displayed to the experimenter as icons (builder, right panel). This makes it straightforward to add new ones.

All you need to do is create a list of parameters that the Component needs to know about (that will automatically appear in the Component's dialog) and a few pieces of code specifying what code should be called at different points in the script (e.g. beginning of the Routine, every frame, end of the study etc...). Many of these will come simply from subclassing the `_base` or `_visual` Components.

To get started, *Add a new feature branch* for the development of this component. (If this doesn't mean anything to you then see *Using the repository*)

You'll mainly be working in the directory `.../psychopy/app/builder/components/`. Take a look at several existing Components (such as `'image.py'`), and key files including `'_base.py'` and `'_visual.py'`.

There are three main steps, the first being by far the most involved.

13.3.1 1. File: newcomp.py

It's pretty straightforward to model a new Component on one of the existing ones. Be prepared to specify what your Component needs to do at several different points in time: before the first trial, every frame, at the end of each routine, and at the end of the experiment. In addition, you may need to sacrifice some complexity in order to keep things streamlined enough for a Builder (see e.g., ratingscale.py).

Your new Component class (in your file *newcomp.py*) should inherit from *BaseComponent* (in *_base.py*), *VisualComponent* (in *_visual.py*), or *KeyboardComponent* (in *keyboard.py*). You may need to rewrite some or all some of these methods, to override default behavior.:

```
class NewcompComponent(BaseComponent): # or (VisualComponent)
    def __init__(...):
        super(NewcompComponent, self).__init__(...)
        ...
    def writeInitCode(self, buff):
    def writeRoutineStartCode(self, buff):
    def writeFrameCode(self, buff):
    def writeRoutineEndCode(self, buff):
```

Calling *super()* will create the basic default set of *params* that almost every component will need: *name*, *startVal*, *startType*, etc. Some of these fields may need to be overridden (e.g., *durationEstim* in *sound.py*). Inheriting from *VisualComponent* (which in turn inherits from *BaseComponent*) adds default visual params, plus arranges for Builder scripts to import *psychopy.visual*. If your component will need other libs, call *self.exp.requirePsychopyLib(['neededLib'])* (see e.g., *parallelPort.py*).

At the top of a component file is a dict named *_localized*. These mappings allow a strict separation of internal string values (= used in logic, never displayed) from values used for display in the Builder interface (= for display only, possibly translated, never used in logic). The *.hint* and *.label* fields of *params['someParam']* should always be set to a localized value, either by using a dict entry such as *_localized['message']*, or via the globally available translation function, *_(‘message’)*. Localized values must **not** be used elsewhere in a component definition.

Very occasionally, you may also need to edit *settings.py*, which writes out the set-up code for the whole experiment (e.g., to define the window). For example, this was necessary for *ApertureComponent*, to pass “allowStencil=True” to the window creation.

Your new Component writes code into a buffer that becomes an executable python file, *xxx_lastrun.py* (where *xxx* is whatever the experimenter specifies when saving from the builder, *xxx.psyexp*). You will do a bunch of this kind of call in your *newcomp.py* file:

```
buff.writeIndented(your_python_syntax_string_here)
```

You have to manage the indentation level of the output code, see *experiment.IndentingBuffer()*.

xxx_lastrun.py is the file that gets built when you run *xxx.psyexp* from the builder. So you will want to look at *xxx_lastrun.py* frequently when developing your component.

Name-space

There are several internal variables (er, names of python objects) that have a specific, hardcoded meaning within *xxx_lastrun.py*. You can expect the following to be there, and they should only be used in the original way (or something will break for the end-user, likely in a mysterious way):

```
'win' = the window
't' = time within the trial loop, referenced to trialClock
'x', 'y' = mouse coordinates, but only if the experimenter uses a mouse component
```

Handling of variable names is under active development, so this list may well be out of date. (If so, you might consider updating it or posting a note to *psychopy-dev*.)

Preliminary testing suggests that there are 600-ish names from *numpy* or *numpy.random*, plus the following:

```
['KeyResponse', '__builtins__', '__doc__', '__file__', '__name__', '__package__', 'buttons', 'core',
```

Yet other names get derived from user-entered names, like trials → thisTrial.

Params

self.params is a key construct that you build up in `__init__`. You need name, startTime, duration, and several other params to be defined or you get errors. ‘name’ should be of type ‘code’.

The Param() class is defined in `psychopy.app.builder.experiment.Param()`. A very useful thing that Params know is how to create a string suitable for writing into the .py script. In particular, the `__str__` representation of a Param will format its value (`.val`) based on its type (`.valType`) appropriately. This means that you don’t need to check or handle whether the user entered a plain string, a string with a code trigger character (\$), or the field was of type `code` in the first place. If you simply request the `str()` representation of the param, it is formatted correctly.

To indicate that a param (eg, `thisParam`) should be considered as an advanced feature, set its category to advanced: `self.params['thisParam'].categ = 'Advanced'`. Then the GUI shown to the experimenter will place it on the ‘Advanced’ tab. Other categories work similarly (`Custom`, etc).

During development, it can sometimes be helpful to save the params into the `xxx_lastrun.py` file as comments, so I could see what was happening:

```
def writeInitCode(self, buff):
    # for debugging during Component development:
    buff.writeIndented("# self.params for aperture:\n")
    for p in self.params.keys():
        try: buff.writeIndented("# %s: %s <type %s>\n" % (p, self.params[p].val, self.params[p].valType))
        except: pass
```

A lot more detail can be inferred from existing components.

Making things loop-compatible looks interesting – see `keyboard.py` for an example, especially code for saving data at the end.

13.3.2 Notes & gotchas

syntax errors in new_comp.py: The PsychoPy app will fail to start if there are syntax error in any of the components that are auto-detected. Just correct them and start the app again.

param[.val]: If you have a boolean variable (e.g., `my_flag`) as one of your params, note that `self.param["my_flag"]` is always True (the param exists → True). So in a boolean context you almost always want the `.val` part, e.g., if `self.param["my_flag"].val`:

However, you do not always want `.val`. Specifically, in a string/unicode context (= to trigger the self-formatting features of Param(s), you almost always want `"%s" % self.param['my_flag']`, without `.val`. Note that it’s better to do this via `"%s"` than `str()` because `str(self.param["my_flag"])` coerces things to type str (squashing unicode) whereas `%s` works for both str and unicode.

13.3.3 2. Icon: newcomp.png

Using your favorite image software, make an icon for your Component with a descriptive name, e.g., ‘newcomp.png’. Dimensions = 48 x 48. Put it in the components directory.

In ‘newcomp.py’, have a line near the top:

```
iconFile = path.join(thisFolder, 'newcomp.png')
```

13.3.4 3. Documentation: newcomp.rst

Just make a descriptively-named text file that ends in `.rst` (“restructured text”), and put it in `psychopy/docs/source/builder/components/`. It will get auto-formatted and end up at <http://www.psychopy.org/builder/components/newcomp.html>

13.4 Style-guide for coder demos

Each coder demo is intended to illustrate a key PsychoPy feature (or two), especially in ways that show usage in practice, and go beyond the description in the API. The aim is not to illustrate every aspect, but to get people up to speed quickly, so they understand how basic usage works, and could then play around with advanced features.

As a newcomer to PsychoPy, you are in a great position to judge whether the comments and documentation are clear enough or not. If something is not clear, you may need to ask a PsychoPy contributor for a description; email psychopy-dev@googlegroups.com.

Here are some style guidelines, written for the OpenHatch event(s) but hopefully useful after that too. These are intended specifically for the coder demos, not for the internal code-base (although they are generally quite close).

The idea is to have clean code that looks and works the same way across demos, while leaving the functioning mostly untouched. Some small changes to function might be needed (e.g., to enable the use of ‘escape’ to quit), but typically only minor changes like this.

- Generally, when you run the demo, does it look good and help you understand the feature? Where might there be room for improvement? You can either leave notes in the code in a comment, or include them in a commit message.
- Standardize the top stuff to have 1) a shbang with python2 (not just python), 2) utf-8 encoding, and 3) a comment:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""Demo name, purpose, description (1-2 sentences, although some demos need more explanation).
"""
```

For the comment / description, it’s a good idea to read and be informed by the relevant parts of the API (see <http://psychopy.org/api/api.html>), but there’s no need to duplicate that text in your comment. If you are unsure, please post to the dev list psychopy-dev@googlegroups.com.

- Follow PEP-8 mostly, some exceptions:
 - current PsychoPy convention is to use camelCase for variable names, so don’t convert those to underscores
 - 80 char columns can spill over a little. Try to keep things within 80 chars most of the time.
 - do allow multiple imports on one line if they are thematically related (e.g., `import os, sys, glob`).
 - inline comments are ok (because the code demos are intended to illustrate and explain usage in some detail, more so than typical code).
- Check all imports:
 - remove any unnecessary ones
 - replace `import time` with `from psychopy import core`. Use `core.getTime()` (= ms since the script started) or `core.getAbsTime()` (= seconds, unix-style) instead of `time.time()`, for all time-related functions or methods not just `time()`.
 - add `from __future__ import division`, even if not needed. And make sure that doing so does not break the demo!
- Fix any typos in comments; convert any lingering British spellings to US, e.g., change *colour* to *color*

- Prefer *if <boolean>*: as a construct instead of *if <boolean> == True*:. (There might not be any to change).
- If you have to choose, opt for more verbose but easier-to-understand code instead of clever or terse formulations. This is for readability, especially for people new to python. If you are unsure, please add a note to your commit message, or post a question to the dev list psychopy-dev@googlegroups.com.
- Standardize variable names:
 - use *win* for the *visual.Window()*, and so *win.flip()*
- Provide a consistent way for a user to exit a demo using the keyboard, ideally enable this on every visual frame: use *if len(event.getKeys(['escape'])): core.quit()*. **Note:** if there is a previous *event.getKeys()* call, it can slurp up the 'escape' keys. So check for 'escape' first.
- Time-out after 10 seconds, if there's no user response and a timeout is appropriate for the demo (and a longer time-out might be needed, e.g., for *ratingScale.py*):

```
demoClock = core.clock() # is demoClock's time is 0.000s at this point
...
if demoClock.getTime() > 10.:
    core.quit()
```

- Most demos are not full screen. For any that are full-screen, see if it can work without being full screen. If it has to be full-screen, add some text to say that pressing 'escape' will quit.
- If displaying log messages to the console seems to help understand the demo, here's how to do it:

```
from psychopy import logging
...
logging.console.setLevel(logging.INFO) # or logging.DEBUG for even more stuff
```

- End a script with *win.close()* (assuming the script used a *visual.Window*), and then *core.quit()* even though it's not strictly necessary

13.5 Localization (I18N, translation)

PsychoPy is used worldwide. Starting with v1.81, many parts of PsychoPy itself (the app) can be translated into any language that has a unicode character set. A translation affects what the experimenter sees while creating and running experiments; it is completely separate from what is shown to the subject. Translations of the online documentation will need a completely different approach.

In the app, translation is handled by a function, `_translate()`, which takes a string argument. (The standard name is `_()`, but unfortunately this conflicts with `_` as used in some external packages that PsychoPy depends on.) The `_translate()` function returns a translated, unicode version of the string in the locale / language that was selected when starting the app. If no translation is available for that locale, the original string is returned (= English).

A locale setting (e.g., 'ja_JP' for Japanese) allows the end-user (= the experimenter) to control the language that will be used for display within the app itself. (It can potentially control other display conventions as well, not just the language.) PsychoPy will obtain the locale from the user preference (if set), or the OS.

Workflow: 1) Make a translation from English (en_US) to another language. You'll need a strong understanding of PsychoPy, English, and the other language. 2) In some cases it will be necessary to adjust PsychoPy's code, but only if new code has been added to the app and that code displays text. Then re-do step 1 to translate the newly added strings.

See notes in `psychopy/app/localization/readme.txt`.

13.5.1 Make a translation (.po file)

As a translator, you will likely introduce many new people to PsychoPy, and your translations will greatly influence their experience. Try to be completely accurate; it is better to leave something in English if you are unsure how PsychoPy is supposed to work.

To translate a given language, you'll need to know the standard 5-character code (see *psychopy/app/localization/mappings*). E.g., for Japanese, wherever LANG appears in the documentation here, you should use the actual code, i.e., "ja_JP" (without quotes).

A free app called poedit is useful for managing a translation. For a given language, the translation mappings (from en_US to LANG) are stored in a .po file (a text file with extension .po); after editing with poedit, these are converted into binary format (with extension .mo) which are used when the app is running.

- Start translation (do these steps once):

Start a translation by opening *psychopy/app/locale/LANG/LC_MESSAGE/messages.po* in Poedit. If there is no such .po file, create a new one:

- make a new directory *psychopy/app/locale/LANG/LC_MESSAGE/* if needed. Your LANG will be auto-detected within PsychoPy only if you follow this convention. You can copy metadata (such as the project name) from another .po file.

Set your name and e-mail address from "Preferences..." of "File" menu. Set translation properties (such as project name, language and charset) from Catalog Properties Dialog, which can be opened from "Properties..." of "Catalog" menu.

In poedit's properties dialog, set the "source keywords" to include '_translate'. This allows poedit to find the strings in PsychoPy that are to be translated.

To add paths where Poedit scans .py files, open "Sources paths" tab on the Catalog Properties Dialog, and set "Base path:" to `../..../..../` (= *psychopy/psychopy/*). Nothing more should be needed. If you've created new catalog, save your catalog to *psychopy/app/locale/LANG/LC_MESSAGE/messages.po*.

Probably not needed, but check anyway: Edit the file containing language code and name mappings, *psychopy/app/localization/mappings*, and fill in the name for your language. Give a name that should be familiar to people who read that language (i.e., use the name of the language as written in the language itself, not in en_US). About 25 are already done.

- Edit a translation:

Open the .po file with Poedit and press "Update" button on the toolbar to update newly added / removed strings that need to be translated. Select a string you want to translate and input your translation to "Translation:" box. If you are unsure where string is used, point on the string in "Source text" box and right-click. You can see where the string is defined.

- Technical terms should not be translated: Builder, Coder, PsychoPy, Flow, Routine, and so on. (See the Japanese translation for guidance.)
- If there are formatting arguments in the original string (`%s, %(first)i`), the same number of arguments must also appear in the translation (but their order is not constrained to be the original order). If they are named (e.g., `%(first)i`), that part should not be translated—here *first* is a python name.
- If you think your translation might have room for improvement, indicate that it is "fuzzy". (Saving Notes does not work for me on Mac, seems like a bug in poedit.)
- After making a new translation, saving it in poedit will save the .po file and also make an associated .mo file. You need to update the .mo file if you want to see your changes reflected in PsychoPy.
- The start-up tips are stored in separate files, and are not translated by poedit. Instead:

- copy the default version (named *psychopy/app/Resources/tips.txt*) to a new file in the same directory, named *tips_LANG.txt*. Then replace English-language tips with translated tips. Note that some of the humor might not translate well, so feel free to leave out things that would be too odd, or include occasional mild humor that would be more appropriate. Humor must be respectful and suitable for using in a classroom, laboratory, or other professional situation. Don't get too creative here. If you have any doubt, best leave it out. (Hopefully it goes without saying that you should avoid any religious, political, disrespectful, or sexist material.)
- in poedit, translate the file name: translate "tips.txt" as "tips_LANG.txt"
- Commit both the .po and .mo files to github (not just one or the other), and any changed files (e.g., *tips_LANG*, *localization/mappings*).

13.5.2 Adjust PsychoPy's code

This is mostly complete (as of 1.81.00), but will be needed for new code that displays text to users of the app (experimenters, not study participants).

There are a few things to keep in mind when working on the app's code to make it compatible with translations. If you are only making a translation, you can skip this section.

- In PsychoPy's code, the language to be used should always be English with American spellings (e.g., "color").
- Within the app, the return value from `_translate()` should be used only for display purposes: in menus, tooltips, etc. A translated value should never be used as part of the logic or internal functioning of PsychoPy. It is purely a "skin". Internally, everything must be in `en_US`.
- Basic usage is exactly what you expect: `_translate("hello")` will return a unicode string at run-time, using mappings for the current locale as provided by a translator in a .mo file. (Not all translations are available yet, see above to start a new one.) To have the app display a translated string to the experimenter, just display the return value from the underscore translation function.
- The strings to be translated must appear somewhere in the app code base as explicit strings within `_translate()`. If you need to translate a variable, e.g., named `str_var` using the expression `_translate(str_var)`, somewhere else you need to explicitly give all the possible values that `str_var` can take, and enclose each of them within the `translate` function. It is okay for that to be elsewhere, even in another file, but not in a comment. This allows poedit to discover of all the strings that need to be translated. (This is one of the purposes of the `_localized` dict at the top of some modules.)
- `_translate()` should not be given a null string to translate; if you use a variable, check that it is not "" to avoid invoking `_translate('')`.
- Strings that contain formatting placeholders (e.g., `%d`, `%s`, `%4f`) require a little more thought. Single placeholders are easy enough: `_translate("hello, %s") % name`.
- Strings with multiple formatting placeholders require named arguments, because positional arguments are not always sufficient to disambiguate things depending on the phrase and the language to be translated into: `_translate("hello, %(first)s %(last)s") % {'first': firstname, 'last': lastname}`
- Localizing drop-down menus is a little more involved. Such menus should display localized strings, but return selected values as integers (`GetSelection()` returns the position within the list). Do not use `GetStringSelection()`, because this will return the localized string, breaking the rule about a strict separation of display and logic. See Builder ParamDialogs for examples.

13.5.3 Other notes

When there are more translations (and if they make the app download large) we might want to manage things differently (e.g., have translations as a separate download from the app).

13.6 Adding a new Menu Item

Adding a new menu-item to the Builder (or Coder) is relatively straightforward, but there are several files that need to be changed in specific ways.

13.6.1 1. makeMenus()

The code that constructs the menus for the Builder is within a method named *makeMenus()*, within class *builder.BuilderFrame()*. Decide which submenu your new command fits under, and look for that section (e.g., File, Edit, View, and so on). For example, to add an item for making the Routine panel items larger, I added two lines within the View menu, by editing the *makeMenus()* method of class *BuilderFrame* within *psychopy/app/builder/builder.py* (similar for Coder):

```
self.viewMenu.Append(self.IDs.tbIncrRoutineSize, _("&Routine Larger\t%s") %self.app.keys['largerRout...
wx.EVT_MENU(self, self.IDs.tbIncrRoutineSize, self.routinePanel.increaseSize)
```

Note the use of the translation function, *_()*, for translating text that will be displayed to users (menu listing, hint).

13.6.2 2. wxIDs.py

A new item needs to have a (numeric) ID so that *wx* can keep track of it. Here, the number is *self.IDs.tbIncrRoutineSize*, which I had to define within the file *psychopy/app/wxIDs.py*:

```
tbIncrRoutineSize=180
```

It's possible that, instead of hard-coding it like this, it's better to make a call to *wx.NewId()* – *wx* will take care of avoiding duplicate IDs, presumably.

13.6.3 3. Key-binding prefs

I also defined a key to use to as a keyboard short-cut for activating the new menu item:

```
self.app.keys['largerRoutine']
```

The actual key is defined in a preference file. Because *psychopy* is multi-platform, you need to add info to four different *.spec* files, all of them being within the *psychopy/preferences/* directory, for four operating systems (Darwin, FreeBSD, Linux, Windows). For *Darwin.spec* (meaning Mac OS X), I added two lines. The first line is not merely a comment: it is also automatically used as a tooltip (in the preferences dialog, under key-bindings), and the second being the actual short-cut key to use:

```
# increase display size of Routines
largerRoutine = string(default='Ctrl++') # on Mac Book Pro this is good
```

This means that the user has to hold down the *Ctrl* key and then press the + key. Note that on Macs, 'Ctrl' in the spec is automatically converted into 'Cmd' for the actual key to use; in the *.spec*, you should always specify things in terms of 'Ctrl' (and not 'Cmd'). The default value is the key-binding to use unless the user defines another one in her or his preferences (which then overrides the default). Try to pick a sensible key for each operating system, and update all four *.spec* files.

13.6.4 4. Your new method

The second line within *makeMenus()* adds the key-binding definition into *wx*'s internal space, so that when the key is pressed, *wx* knows what to do. In the example, it will call the method *self.routinePanel.increaseSize*, which I had to

define to do the desired behavior when the method is called (in this case, increment an internal variable and redraw the routine panel at the new larger size).

13.6.5 5. Documentation

To let people know that your new feature exists, add a note about your new feature in the CHANGELOG.txt, and appropriate documentation in .rst files.

Happy Coding Folks!!

PsychoPy Experiment file format (.psyexp)

The file format used to save experiments constructed in PsychoPy builder was created especially for the purpose, but is an open format, using a basic xml form, that may be of use to other similar software. Indeed the builder itself could be used to generate experiments on different backends (such as Vision Egg, PsychToolbox or PyEPL). The xml format of the file makes it extremely platform independent, as well as moderately(?) easy to read by humans. There was a further suggestion to generate an XSD (or similar) [schema against which psyexp files could be validated](#). That is a low priority but welcome addition if you wanted to work on it(!) There is a basic XSD (XML Schema Definition) available in *psychopy/app/builder/experiment.xsd*.

The simplest way to understand the file format is probably simply to create an experiment, save it and open the file in an xml-aware editor/viewer (e.g. change the file extension from .psyexp to .xml and then open it in Firefox). An example (from the stroop demo) is shown below.

The file format maps fairly obviously onto the structure of experiments constructed with the *Builder* interface, as described [here](#). There are general *Settings* for the experiment, then there is a list of *Routines* and a *Flow* that describes how these are combined.

As with any xml file the format contains object *nodes* which can have direct properties and also child nodes. For instance the outermost node of the .psyexp file is the experiment node, with properties that specify the version of PsychoPy that was used to save the file most recently and the encoding of text within the file (ascii, unicode etc.), and with child nodes *Settings*, *Routines* and *Flow*.

14.1 Parameters

Many of the nodes described within this xml description of the experiment contain Param entries, representing different parameters of that Component. Nearly all parameter nodes have a *name* property and a *val* property. The parameter node with the name “advancedParams” does not have them. Most also have a *valType* property, which can take values ‘bool’, ‘code’, ‘extendedCode’, ‘num’, ‘str’ and an *updates* property that specifies whether this parameter is changing during the experiment and, if so, whether it changes ‘every frame’ (of the monitor) or ‘every repeat’ (of the Routine).

14.2 Settings

The Settings node contains a number of parameters that, in PsychoPy, would normally be set in the *Experiment settings* dialog, such as the monitor to be used. This node contains a number of *Parameters* that map onto the entries in that dialog.

14.3 Routines

This node provides a sequence of xml child nodes, each of which describes a *Routine*. Each Routine contains a number of children, each specifying a *Component*, such as a stimulus or response collecting device. In the *Builder* view, the *Routines* obviously show up as different tabs in the main window and the *Components* show up as tracks within that tab.

14.4 Components

Each *Component* is represented in the .psyexp file as a set of parameters, corresponding to the entries in the appropriate component dialog box, that completely describe how and when the stimulus should be presented or how and when the input device should be read from. Different *Components* have slightly different nodes in the xml representation which give rise to different sets of parameters. For instance the *TextComponent* nodes has parameters such as *colour* and *font*, whereas the *KeyboardComponent* node has parameters such as *forceEndTrial* and *correctIf*.

14.5 Flow

The Flow node is rather more simple. Its children simply specify objects that occur in a particular order in time. A Routine described in this flow must exist in the list of Routines, since this is where it is fully described. One Routine can occur once, more than once or not at all in the Flow. The other children that can occur in a Flow are LoopInitiators and LoopTerminators which specify the start and endpoints of a loop. All loops must have exactly one initiator and one terminator.

14.6 Names

For the experiment to generate valid PsychoPy code the name parameters of all objects (Components, Loops and Routines) must be unique and contain no spaces. That is, an experiment can not have two different Routines called 'trial', nor even a Routine called 'trial' and a Loop called 'trial'.

The Parameter names belonging to each Component (or the Settings node) must be unique within that Component, but can be identical to parameters of other Components or can match the Component name themselves. A TextComponent should not, for example, have multiple 'pos' parameters, but other Components generally will, and a Routine called 'pos' would also be also permissible.

```
<PsychoPy2experiment version="1.50.04" encoding="utf-8">
  <Settings>
    <Param name="Monitor" val="testMonitor" valType="str" updates="None"/>
    <Param name="Window size (pixels)" val="[1024, 768]" valType="code" updates="None"/>
    <Param name="Full-screen window" val="True" valType="bool" updates="None"/>
    <Param name="Save log file" val="True" valType="bool" updates="None"/>
    <Param name="Experiment info" val="{ 'participant': 's_001', 'session': 001 }" valType="code" updates="None"/>
    <Param name="Show info dlg" val="True" valType="bool" updates="None"/>
    <Param name="logging level" val="warning" valType="code" updates="None"/>
    <Param name="Units" val="norm" valType="str" updates="None"/>
    <Param name="Screen" val="1" valType="num" updates="None"/>
  </Settings>
  <Routines>
    <Routine name="trial">
      <TextComponent name="word">
        <Param name="name" val="word" valType="code" updates="constant"/>
      </TextComponent>
    </Routine>
  </Routines>
</PsychoPy2experiment>
```



```

    </Routine>
</Routines>
<Flow>
    <Routine name="instruct"/>
    <LoopInitiator loopType="TrialHandler" name="trials">
        <Param name="endPoints" val="[0, 1]" valType="num" updates="None"/>
        <Param name="name" val="trials" valType="code" updates="None"/>
        <Param name="loopType" val="random" valType="str" updates="None"/>
        <Param name="nReps" val="5" valType="num" updates="None"/>
        <Param name="trialList" val="[{ 'text': 'red', 'rgb': [1, -1, -1], 'congruent': 1, 'corrAns': 1" updates="None"
        <Param name="trialListFile" val="/Users/jwp...troop/trialTypes.csv" valType="str" updates="None"
    </LoopInitiator>
    <Routine name="trial"/>
    <LoopTerminator name="trials"/>
    <Routine name="thanks"/>
</Flow>
</PsychoPy2experiment>

```


p

- `psychopy.core`, [83](#)
- `psychopy.data`, [91](#)
- `psychopy.hardware.egi`, [112](#)
- `psychopy.hardware.minolta`, [113](#)
- `psychopy.hardware.pr`, [115](#)
- `psychopy.iohub.client`, [120](#)
- `psychopy.iohub.client.keyboard`, [120](#)
- `psychopy.logging`, [121](#)
- `psychopy.misc`, [124](#)
- `psychopy.parallel`, [131](#)
- `psychopy.preferences`, [132](#)
- `psychopy.sound`, [133](#)
- `psychopy.tools`, [133](#)
- `psychopy.tools.colorspectools`, [133](#)
- `psychopy.tools.coordinatetools`, [135](#)
- `psychopy.tools.filetools`, [135](#)
- `psychopy.tools.imagetools`, [136](#)
- `psychopy.tools.monitorunittools`, [137](#)
- `psychopy.tools.plottools`, [138](#)
- `psychopy.tools.typetools`, [138](#)
- `psychopy.tools.unittools`, [138](#)

A

abort() (psychopy.data.ExperimentHandler method), 92
 Adaptive staircase, 28
 add() (psychopy.core.Clock method), 84
 addData() (psychopy.data.ExperimentHandler method), 92
 addData() (psychopy.data.MultiStairHandler method), 100
 addData() (psychopy.data.QuestHandler method), 104
 addData() (psychopy.data.StairHandler method), 97
 addData() (psychopy.data.TrialHandler method), 94
 addLevel() (in module psychopy.logging), 122
 addLoop() (psychopy.data.ExperimentHandler method), 92
 addOtherData() (psychopy.data.MultiStairHandler method), 100
 addOtherData() (psychopy.data.QuestHandler method), 104
 addOtherData() (psychopy.data.StairHandler method), 97
 addResponse() (psychopy.data.MultiStairHandler method), 100
 addResponse() (psychopy.data.QuestHandler method), 104
 addResponse() (psychopy.data.StairHandler method), 97
 array2image() (in module psychopy.tools.imagetools), 137

B

bootStraps() (in module psychopy.data), 109

C

calculateNextIntensity() (psychopy.data.QuestHandler method), 104
 calculateNextIntensity() (psychopy.data.StairHandler method), 97
 cart2pol() (in module psychopy.tools.coordinatetools), 135
 cart2sph() (in module psychopy.tools.coordinatetools), 135
 checkOK() (psychopy.hardware.minolta.LS100 method), 114

clearMemory() (psychopy.hardware.minolta.LS100 method), 114
 Clock (class in psychopy.core), 83
 cm2deg() (in module psychopy.tools.monitorunittools), 137
 cm2pix() (in module psychopy.tools.monitorunittools), 137
 complete() (psychopy.core.StaticPeriod method), 85
 confInterval() (psychopy.data.QuestHandler method), 104
 convertToPix() (in module psychopy.tools.monitorunittools), 137
 copyCalib() (psychopy.monitors.Monitor method), 126
 CountdownTimer (class in psychopy.core), 84
 CPU, 28
 critical() (in module psychopy.logging), 122
 CRT, 28
 csv, 28

D

data() (in module psychopy.logging), 122
 debug() (in module psychopy.logging), 122
 deg2cm() (in module psychopy.tools.monitorunittools), 137
 deg2pix() (in module psychopy.tools.monitorunittools), 137
 degrees() (in module psychopy.tools.unittools), 139
 delCalib() (psychopy.monitors.Monitor method), 126
 dkl2rgb() (in module psychopy.tools.colorspectools), 134
 dklCart2rgb() (in module psychopy.tools.colorspectools), 134

E

endRemoteMode() (psychopy.hardware.pr.PR655 method), 116
 error() (in module psychopy.logging), 122
 eval() (psychopy.data.FitCumNormal method), 108
 eval() (psychopy.data.FitLogistic method), 107
 eval() (psychopy.data.FitNakaRushton method), 107
 eval() (psychopy.data.FitWeibull method), 106

exp() (in module psychopy.logging), 122
 ExperimentHandler (class in psychopy.data), 91

F

fatal() (in module psychopy.logging), 122
 findPhotometer() (in module psychopy.hardware), 118
 findPR650() (in module psychopy.monitors), 129
 FitCumNormal (class in psychopy.data), 107
 fitGammaErrFun() (psychopy.monitors.GammaCalculator method), 129
 fitGammaFun() (psychopy.monitors.GammaCalculator method), 129
 FitLogistic (class in psychopy.data), 107
 FitNakaRushton (class in psychopy.data), 107
 FitWeibull (class in psychopy.data), 106
 float_uint16() (in module psychopy.tools.typetools), 138
 float_uint8() (in module psychopy.tools.typetools), 138
 flush() (in module psychopy.logging), 122, 123
 fromFile() (in module psychopy.tools.filetools), 135
 functionFromStaircase() (in module psychopy.data), 108

G

GammaCalculator (class in psychopy.monitors), 129
 gammaFun() (in module psychopy.monitors), 130
 gammaInvFun() (in module psychopy.monitors), 130
 gammalsDefault() (psychopy.monitors.Monitor method), 126
 genDelimiter() (in module psychopy.tools.filetools), 136
 getAbsTime() (in module psychopy.core), 83
 getAllMonitors() (in module psychopy.monitors), 129
 getCalibDate() (psychopy.monitors.Monitor method), 126
 getDeviceSN() (psychopy.hardware.pr.PR655 method), 116
 getDeviceType() (psychopy.hardware.pr.PR655 method), 116
 getDistance() (psychopy.monitors.Monitor method), 126
 getDKL_RGB() (psychopy.monitors.Monitor method), 126
 getEarlierTrial() (psychopy.data.TrialHandler method), 94
 getExp() (psychopy.data.MultiStairHandler method), 100
 getExp() (psychopy.data.QuestHandler method), 104
 getExp() (psychopy.data.StairHandler method), 97
 getExp() (psychopy.data.TrialHandler method), 94
 getFutureTrial() (psychopy.data.TrialHandler method), 94
 getGamma() (psychopy.monitors.Monitor method), 126
 getGammaGrid() (psychopy.monitors.Monitor method), 126
 getLastColorTemp() (psychopy.hardware.pr.PR655 method), 116
 getLastLum() (psychopy.hardware.pr.PR650 method), 115

getLastResetTime() (psychopy.core.MonotonicClock method), 84
 getLastSpectrum() (psychopy.hardware.pr.PR650 method), 115
 getLastSpectrum() (psychopy.hardware.pr.PR655 method), 116
 getLastTristim() (psychopy.hardware.pr.PR655 method), 117
 getLastUV() (psychopy.hardware.pr.PR655 method), 117
 getLastXY() (psychopy.hardware.pr.PR655 method), 117
 getLevel() (in module psychopy.logging), 122
 getLevelsPost() (psychopy.monitors.Monitor method), 127
 getLevelsPre() (psychopy.monitors.Monitor method), 127
 getLinearizeMethod() (psychopy.monitors.Monitor method), 127
 getLMS_RGB() (psychopy.monitors.Monitor method), 126
 getLum() (psychopy.hardware.minolta.LS100 method), 114
 getLum() (psychopy.hardware.pr.PR650 method), 115
 getLumSeriesPR650() (in module psychopy.monitors), 129
 getLumsPost() (psychopy.monitors.Monitor method), 127
 getLumsPre() (psychopy.monitors.Monitor method), 127
 getMeanLum() (psychopy.monitors.Monitor method), 127
 getNotes() (psychopy.monitors.Monitor method), 127
 getOriginPathAndFile() (psychopy.data.MultiStairHandler method), 100
 getOriginPathAndFile() (psychopy.data.QuestHandler method), 104
 getOriginPathAndFile() (psychopy.data.StairHandler method), 98
 getOriginPathAndFile() (psychopy.data.TrialHandler method), 94
 getPsychopyVersion() (psychopy.monitors.Monitor method), 127
 getRGBspectra() (in module psychopy.monitors), 130
 getSizePix() (psychopy.monitors.Monitor method), 127
 getSpectra() (psychopy.monitors.Monitor method), 127
 getSpectrum() (psychopy.hardware.pr.PR650 method), 115
 getTime() (in module psychopy.core), 83
 getTime() (psychopy.core.CountdownTimer method), 84
 getTime() (psychopy.core.MonotonicClock method), 84
 getUseBits() (psychopy.monitors.Monitor method), 127
 getWidth() (psychopy.monitors.Monitor method), 127
 GPU, 28

H

haveInternetAccess() (in module psychopy.web), 140
 hsv2rgb() (in module psychopy.tools.colorspectools), 134

I

image2array() (in module psychopy.tools.imagetools), 137
importConditions() (in module psychopy.data), 108
importData() (psychopy.data.QuestHandler method), 104
incTrials() (psychopy.data.QuestHandler method), 104
info() (in module psychopy.logging), 123
inverse() (psychopy.data.FitCumNormal method), 108
inverse() (psychopy.data.FitLogistic method), 107
inverse() (psychopy.data.FitNakaRushton method), 107
inverse() (psychopy.data.FitWeibull method), 106

L

lineariseLums() (psychopy.monitors.Monitor method), 127
lms2rgb() (in module psychopy.tools.colorspectools), 134
loadAll() (psychopy.preferences.Preferences method), 132
loadUserPrefs() (psychopy.preferences.Preferences method), 132
log() (in module psychopy.logging), 123
LogFile (class in psychopy.logging), 122
loopEnded() (psychopy.data.ExperimentHandler method), 92
LS100 (class in psychopy.hardware.minolta), 113

M

makeDKL2RGB() (in module psychopy.monitors), 130
makeImageAuto() (in module psychopy.tools.imagetools), 137
makeLMS2RGB() (in module psychopy.monitors), 130
mean() (psychopy.data.QuestHandler method), 104
measure() (psychopy.hardware.minolta.LS100 method), 114
measure() (psychopy.hardware.pr.PR650 method), 116
measure() (psychopy.hardware.pr.PR655 method), 117
mergeFolder() (in module psychopy.tools.filetools), 136
Method of constants, 28
mode() (psychopy.data.QuestHandler method), 104
Monitor (class in psychopy.monitors), 126
MonotonicClock (class in psychopy.core), 84
MultiStairHandler (class in psychopy.data), 99

N

newCalib() (psychopy.monitors.Monitor method), 127
next() (psychopy.data.MultiStairHandler method), 100
next() (psychopy.data.QuestHandler method), 105
next() (psychopy.data.StairHandler method), 98
next() (psychopy.data.TrialHandler method), 94
nextEntry() (psychopy.data.ExperimentHandler method), 92

O

openOutputFile() (in module psychopy.tools.filetools), 136

P

ParallelPort (in module psychopy.parallel), 131
parseSpectrumOutput() (psychopy.hardware.pr.PR650 method), 116
parseSpectrumOutput() (psychopy.hardware.pr.PR655 method), 117
pix2cm() (in module psychopy.tools.monitorunittools), 138
pix2deg() (in module psychopy.tools.monitorunittools), 138
plotFrameIntervals() (in module psychopy.tools.plottools), 138
pol2cart() (in module psychopy.tools.coordinatetools), 135
PR650 (class in psychopy.hardware.pr), 115
PR655 (class in psychopy.hardware.pr), 116
Preferences (class in psychopy.preferences), 132
printAsText() (psychopy.data.MultiStairHandler method), 101
printAsText() (psychopy.data.QuestHandler method), 105
printAsText() (psychopy.data.StairHandler method), 98
printAsText() (psychopy.data.TrialHandler method), 94
psychopy.core (module), 83
psychopy.data (module), 91
psychopy.hardware.egi (module), 112
psychopy.hardware.minolta (module), 113
psychopy.hardware.pr (module), 115
psychopy.iohub.client (module), 120
psychopy.iohub.client.keyboard (module), 120
psychopy.logging (module), 121
psychopy.misc (module), 124
psychopy.parallel (module), 131
psychopy.preferences (module), 132
psychopy.sound (module), 133
psychopy.tools (module), 133
psychopy.tools.colorspectools (module), 133
psychopy.tools.coordinatetools (module), 135
psychopy.tools.filetools (module), 135
psychopy.tools.imagetools (module), 136
psychopy.tools.monitorunittools (module), 137
psychopy.tools.plottools (module), 138
psychopy.tools.typetools (module), 138
psychopy.tools.unittools (module), 138

Q

quantile() (psychopy.data.QuestHandler method), 105
QuestHandler (class in psychopy.data), 102

R

radians() (in module psychopy.tools.unittools), 138

readPin() (psychopy.parallel method), 132
 requireInternetAccess() (in module psychopy.web), 140
 reset() (psychopy.core.Clock method), 84
 resetPrefs() (psychopy.preferences.Preferences method), 132
 restoreBadPrefs() (psychopy.preferences.Preferences method), 132
 rgb2dkiCart() (in module psychopy.tools.colorspectools), 134
 rgb2lms() (in module psychopy.tools.colorspectools), 134

S

saveAppData() (psychopy.preferences.Preferences method), 132
 saveAsExcel() (psychopy.data.MultiStairHandler method), 101
 saveAsExcel() (psychopy.data.QuestHandler method), 105
 saveAsExcel() (psychopy.data.StairHandler method), 98
 saveAsExcel() (psychopy.data.TrialHandler method), 94
 saveAsPickle() (psychopy.data.ExperimentHandler method), 92
 saveAsPickle() (psychopy.data.MultiStairHandler method), 101
 saveAsPickle() (psychopy.data.QuestHandler method), 105
 saveAsPickle() (psychopy.data.StairHandler method), 99
 saveAsPickle() (psychopy.data.TrialHandler method), 95
 saveAsText() (psychopy.data.MultiStairHandler method), 101
 saveAsText() (psychopy.data.QuestHandler method), 106
 saveAsText() (psychopy.data.StairHandler method), 99
 saveAsText() (psychopy.data.TrialHandler method), 95
 saveAsWideText() (psychopy.data.ExperimentHandler method), 92
 saveAsWideText() (psychopy.data.TrialHandler method), 96
 saveMon() (psychopy.monitors.Monitor method), 127
 saveUserPrefs() (psychopy.preferences.Preferences method), 133
 sd() (psychopy.data.QuestHandler method), 106
 sendMessage() (psychopy.hardware.minolta.LS100 method), 114
 sendMessage() (psychopy.hardware.pr.PR650 method), 116
 sendMessage() (psychopy.hardware.pr.PR655 method), 117
 setCalibDate() (psychopy.monitors.Monitor method), 127
 setCurrent() (psychopy.monitors.Monitor method), 127
 setData() (psychopy.parallel method), 131
 setDefaultClock() (in module psychopy.logging), 123
 setDistance() (psychopy.monitors.Monitor method), 128

setDKL_RGB() (psychopy.monitors.Monitor method), 128
 setExp() (psychopy.data.MultiStairHandler method), 102
 setExp() (psychopy.data.QuestHandler method), 106
 setExp() (psychopy.data.StairHandler method), 99
 setExp() (psychopy.data.TrialHandler method), 96
 setGamma() (psychopy.monitors.Monitor method), 128
 setGammaGrid() (psychopy.monitors.Monitor method), 128
 setLevel() (psychopy.logging.LogFile method), 122
 setLevelsPost() (psychopy.monitors.Monitor method), 128
 setLevelsPre() (psychopy.monitors.Monitor method), 128
 setLineariseMethod() (psychopy.monitors.Monitor method), 128
 setLMS_RGB() (psychopy.monitors.Monitor method), 128
 setLumsPost() (psychopy.monitors.Monitor method), 128
 setLumsPre() (psychopy.monitors.Monitor method), 128
 setMaxAttempts() (psychopy.hardware.minolta.LS100 method), 114
 setMeanLum() (psychopy.monitors.Monitor method), 128
 setMode() (psychopy.hardware.minolta.LS100 method), 114
 setNotes() (psychopy.monitors.Monitor method), 128
 setPin() (psychopy.parallel method), 131
 setPortAddress() (psychopy.parallel method), 131
 setPsychopyVersion() (psychopy.monitors.Monitor method), 128
 setSizePix() (psychopy.monitors.Monitor method), 128
 setSpectra() (psychopy.monitors.Monitor method), 128
 setupProxy() (in module psychopy.web), 141
 setUseBits() (psychopy.monitors.Monitor method), 128
 setWidth() (psychopy.monitors.Monitor method), 128
 simulate() (psychopy.data.QuestHandler method), 106
 sph2cart() (in module psychopy.tools.coordinatestools), 135
 StairHandler (class in psychopy.data), 96
 start() (psychopy.core.StaticPeriod method), 85
 startRemoteMode() (psychopy.hardware.pr.PR655 method), 117
 StaticPeriod (class in psychopy.core), 84

T

toFile() (in module psychopy.tools.filetools), 135
 TrialHandler (class in psychopy.data), 93

U

uint8_float() (in module psychopy.tools.typetools), 138
 upload() (in module psychopy.web), 140

V

validate() (psychopy.preferences.Preferences method),

133

VBI, [28](#)

VBI blocking, [28](#)

VBI syncing, [28](#)

W

wait() (in module psychopy.core), [83](#)

warn() (in module psychopy.logging), [123](#)

warning() (in module psychopy.logging), [123](#)

write() (psychopy.logging.LogFile method), [122](#)

X

xlsx, [28](#)