Chapter 2 Building experiments with PsychoPy

There are quite a few Python libraries that one can use to build psychological experiments. PsychoPy, which also works as a standalone application (on Windows and macOS), is one of the most popular. The low-level libraries that PsychoPy heavily depends on are Pygame, Pyglet, and a few other libraries for image manipulation and movie and audio playback. Building upon those libraries is a set of convenient functions for generating complex visual stimuli (e.g., checkerboard and Gabor patch), for interfacing with keyboards and mouse, and interfacing with eye-trackers, EEG, and other frequently used research equipment. PsychoPy also includes handy tools for monitor calibration, color conversion, etc. The only downside of PsychoPy is that it depends on many libraries, and these dependencies may occasionally break down.

As an actively maintained and rapidly evolving research tool, it is reasonable to see that some old features become deprecated, and new features become available in the development circle of PsychoPy. This chapter will cover the essential functions that were active at the time of writing and are likely to remain active in the immediate future. I will first introduce a few modules of PsychoPy (e.g., visual, monitors, events), with examples, of course; then, I will present a simple method for handling experimental trials.

Install PsychoPy

PsychoPy can be installed as a Python module with "*pip install psychopy*" on the command line on Windows, or "*sudo pip install psychopy*" on macOS. On Ubuntu, the command we use to install PsychoPy is "*sudo apt-get install python-psychopy*". If a graphical user interface is preferred, you can also install

the Standalone version of PsychoPy on Windows or macOS. The standalone version comes with Python 3.6; it is a self-contained Python environment that will not interfere with the other versions of Python installed on your machine.

The standalone version of PsychoPy has a graphical programming interface—known as "Builder"—for building experimental tasks through dragging and dropping functional components onto a timeline. This book will not cover the Builder interface. Instead, we will focus on the "Coder" interface of PsychoPy, which allows users to write and debug scripts as if PsychoPy is just another Python IDE.

At the time this book was being written, the PsychoPy version was 3.2, https://www.psychopy.org/. Nevertheless, the example scripts included in this book should also work in newer releases.

PsychoPy Coder

PsychoPy comes with a simple IDE (Coder) that allows users to build tasks through scripting. The PsychoPy Coder interface consists of an editor, a Python shell, and an Output window for presenting debugging information, e.g., warnings, errors, and the output of a script (see Figure 2-1). One can hide or show the Output/Shell window by toggling the "View-> Show Output/Shell" menu entry.

We will cover the various features of the PsychoPy modules in detail in later sections of this chapter. Let us first go through a short script to get to know the essential elements of PsychoPy. This short script requires the *visual, core, event, and monitor* modules, and it will show a drifting Gabor on the screen until you press a key. To run the script, save it with a *.py* extension and then click the Run button on the menu bar or press the hotkey F5. This short script does not rely on PsychoPy Coder; it is just a Python script that one can open and edit in IDLE or any other IDE or script editor.

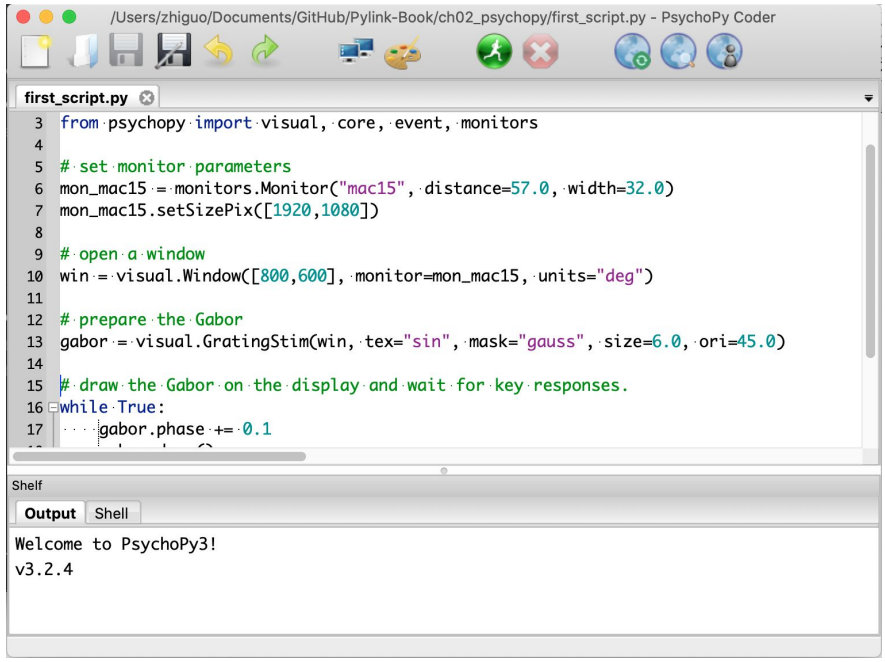

Figure 2-1 *The Coder interface of PsychoPy.*

```
# Filename: first_script.py

from psychopy import visual, core, event, monitors

# set monitor parameters
mon_mac15 = monitors.Monitor("mac15", distance=57.0, width=32.0)
mon_mac15.setSizePix([1920,1080])

# open a window
win = visual.Window([800,600], monitor=mon_mac15, units="deg")

# prepare the Gabor
gabor = visual.GratingStim(win, tex="sin", mask="gauss", size=6.0, ori=45.0)

# draw the Gabor on the display and wait for key responses.
while True:
    gabor.phase += 0.1
    gabor.draw()
    win.flip()
    key = event.getKeys()
    if len(key)>0: # press any key to quit
```

```
        win.close()
        core.quit()
```

The structure of this script is fairly straightforward. First, we import the PsyhoPy modules needed for this script. Then, we instantiate a Monitor object with *monitor.Monitor*(). We need a Monitor object to store the various monitor parameters, e.g., eye-to-screen distance (in cm), screen width (in cm), and the native screen resolution (in pixels).

```
mon_mac15 = monitors.Monitor("mac15", distance=57.0, width=32.0)
mon_mac15.setSizePix([1920,1080])
```

With the correct monitor parameters, we can then open a Window by calling *visual.Window*().

```
win = visual.Window([800,600], monitor=mon_mac15, units="deg")
```

One of the advantages of PsychoPy is that it allows users to manipulate the properties of the various objects dynamically. We first initialize the visual stimuli in memory and then change their properties (e.g., position, color, spatial frequency, phase, etc.) on the fly during testing. In this script, we first instantiate a Gabor patch with *visual.GratingStim()* in the memory. We set the size of the Gabor patch to 6.0 degrees of visual angle (we will discuss screen units later) and the orientation to 45.0 degrees. Note that the 0-degree direction in PsychoPy is the 90-degree direction in the polar coordinates, and the values increase in the clockwise direction.

```
gabor = visual.GratingStim(win, tex="sin", mask="gauss", size=6.0, ori=45.0)
```

The above line of code creates a *visual.GratingStim*() object in memory. The *draw*() function will prepare the stimuli in the back buffer in video memory; the *flip*() function will flip the back buffer to the front to show the stimuli on the screen.[1] We repeat these operations in a *while*-loop. Note that the phase of the

[1] Modern video graphics cards all support hardware double buffering. At any one time, the contents in the front buffer is actively being displayed on the screen, while the back buffer is being drawn. When the drawing is complete in the back buffer, the two buffers can be switched (flipped) so the back buffer becomes the front buffer and its content is shown on screen, usually in sync with the monitor vertical blanking signal.

Gabor is constantly updated, by adding 0.1,[2] before we call *draw()*. So, the Gabor will drift until the while-loop terminates.

```
while True:
    gabor.phase += 0.1
    gabor.draw()
    win.flip()
    key = event.getKeys()
    if len(key)>0:  # press any key to quit
        core.quit()
```

This short script also shows a way of checking keyboard presses. The *event.getKeys*() function will return a list of the keys you have pressed since the last call of this function. We would terminate the PsychoPy script all together with *core.quit()* if the list (of keys) returned by*event.getKeys*() is not empty.

Open a window

A window is required before we draw any visual stimulus on the screen. A PsychoPy window is just like any other computer application window; you can resize it and move it around. As shown in the short script in the previous section, opening a window involves only one line of code. However, there are parameters that one needs to consider when opening a window. Here we will skip the straightforward ones (e.g., size, pos, color, etc.).

```
visual.Window(size=(800, 600), pos=None, color=(0, 0, 0), colorSpace='rgb',
rgb=None, dkl=None, lms=None, fullscr=None, allowGUI=None, monitor=None,
bitsMode=None, winType=None, units=None, gamma=None, blendMode='avg',
screen=0, viewScale=None, viewPos=None, viewOri=0.0, waitBlanking=True,
allowStencil=False, multiSample=False, numSamples=2, stereo=False,
name='window1', checkTiming=True, useFBO=False, useRetina=True,
autoLog=True, *args, **kwargs)
```

Screen units

The screen unit in computer graphics is a “pixel”. In Pygame, the pixel coordinate of the top-left corner of the screen is (0, 0), while the bottom-right corner of the screen corresponds to the maximum number of horizontal and

[2] One of the quirky bits of PsychoPy is that *phase* has modulus 1, rather than 360 or 2*pi.

vertical pixels minus 1. One departure of PsychoPy from this convention is that the origin of the screen coordinates is the center of the screen, with negative values running leftwards and downwards and positive values running rightwards and upwards, as in the Cartesian coordinates.

In addition to screen pixels, PsychoPy also supports several other screen units. The most useful screen unit in vision science is "visual angle" (in degrees), which reflects the size of an object on the retina (see Figure 2-2 for an illustration).

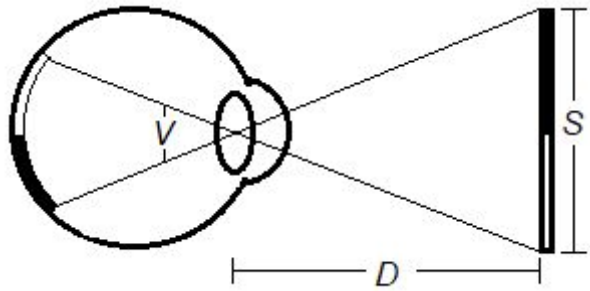

Figure 2-2 *Visual angle calculation. In this illustration, the visual angle (V) for object S is 2 * arctan(S/2D).*

Pixels are the native units in computer graphics. To draw a line that is 1 degree (visual angle) long on the monitor, we need to derive how many pixels correspond to 1 degree of visual angle, based on the resolution of the screen (in pixels), the physical size of the visible screen area (in centimeters), and the eye-to-screen distance (in centimeters). Things can be a bit more complicated if we consider the fact that most computer monitors have flat screens and the eye-to-screen distance varies with the position on the screen. In PsychoPy, you can use one of the following screen degree units:

- *deg* –assumes one degree of visual angle spans the same number of pixels at all parts of the screen
- *degFlatPos*—corrects screen flatness only for stimuli position, no change in stimulus size and shape
- *degFlat*—corrects screen flatness for each vertex of the stimuli so that a square will get larger and rhomboid in the peripheral

In addition to degrees of visual angle, PsychoPy also supports the following screen units:

- *norm*—normalized, screen ranges from -1 to 1 in both x and y
- *cm*—centimeters on screen
- *height*—relative to the height of the screen. The vertical screen range is -0.5 to 0.5; for a 4:3 ratio screen, the horizontal screen range is -0.67 to 0.67.
- *pix*—screen pixel

In my experience, 'norm', 'cm', and 'height' are rarely used. Screen pixel ('pix') is what I would recommend for eye-tracking studies, as the gaze position returned by the tracker is also in screen pixel coordinates. Please bear in mind that the origin of the screen pixel coordinates in PsychoPy is the center of the screen, whereas the EyeLink tracker uses the computer graphics convention, i.e., the origin is the top-left corner of the screen, with positive values running rightwards and downwards.

Monitor

To correctly present stimuli in the required screen unit (e.g., degree of visual angle), PsychoPy requires some information about the monitor. For instance, the viewing distance and the physical size and resolution of the screen. I strongly recommend users to set up a Monitor object to store these parameters explicitly.

The code below shows how to create a *Monitor* object. The *name* of this monitor is "myMac15", and the screen width and viewing distance are 32 and 57 cm, respectively. We then set the resolution of the screen to 1024 x 768, by calling *setSizePix*().

The parameters of a *monitor* object can be saved in a *json* configuration file, by calling *save*() or by using the graphical interface in Coder (Monitor center). You can find the *json* files in the "monitors" subfolder of your PsychoPy configuration folder. On Linux and macOS, the path to the monitor

configuration files is '~/.psychopy3/monitors';[3] on Windows, the path to the configuration files is '%APPDATA%/psychopy3/monitors'.[4]

```
mon_mac15 = monitors.Monitor("mac15", distance=57.0, width=32.0)
mon_mac15.setSizePix([1920,1080])
mon_mac15.save()
```

On my macOS, I have two configuration files, the default 'testMonitor.json' and the one generated by the above lines of code.

```
Zhiguos-MacBook-Pro-2:~ zhiguo$ ls .psychopy3/monitors/
mac15.json            testMonitor.json
```

When opening a new window, you can pass either a monitor object (e.g., mon_mac15) or the name of the *json* configuration file (e.g., 'mac15') to the "monitor" argument. The former is what I would recommend because it improves code transparency.

```
win = visual.Window([800,600], monitor=mon_mac15, units="deg")
```

If the 'monitor' argument is left out, PsychoPy may use the incorrect parameters to estimate how many pixels correspond to one degree of visual angle. The key message here is not to use 'deg' as the screen unit if you do not have a properly configured monitor – stimuli may well end up in the wrong sizes.

Window types

PsychoPy uses several low-level libraries to manage the window (graphics context). The default is "pyglet", but one can use a different backend by setting the "winType" parameter to either "pyglet", "pygame", or "glfw". Pygame is officially deprecated, but it may still work in PsychoPy 3. The default is Pyglet because it allows managing multiple windows and rendering movies. The

[3] On Linux, MacOS, and other Unix or Unix-like operating systems, the pre-fix period (".") of a file or folder means that a file or folder is hidden, invisible in a file browser. The tilde ('~') represents the home folder of a user.

[4] On Window, '%APPDATA%' is a system variable that represents the path to the user AppData folder. For instance, typing %APPDATA% in the address bar of File Explorer will bring me to C:\Users\Eyelink\AppData\Roaming, where 'EyeLink' is my user folder. The full path to the monitor configuration .json file on my Windows machine would be C:\Users\Eyelink\AppData\Roaming\psychopy3\monitors.

GLFW backend is a new addition that allows more flexible window configurations.

Gamma

The term gamma comes from the third letter of the Greek alphabet, written Γ in upper case, and γ in lower case. In computer graphics, the output device (e.g., a monitor or printer) does not always give the requested output. For instance, the RGB values for black and white are (0, 0, 0) and (255, 255, 255). You may imagine that increasing the values in the RGB triplets from 0 to 255 will give you a grayscale of linearly increasing brightness. Your assumption holds in the video memory, but when these RGB triplets are sent to the monitor to process, the resulting grayscale on the screen is not linear and follows a gamma function (in the simplest format, $y = x^{\gamma}$). To show this phenomenon in a real-world example, we measured the luminance of a dot on an LCD screen with a photometer, while linearly increasing the RGB values from 0 -255. The results are plotted in Figure 2-3, which reveals the non-linear increase of luminance (i.e., γ does not equal to 1.0).

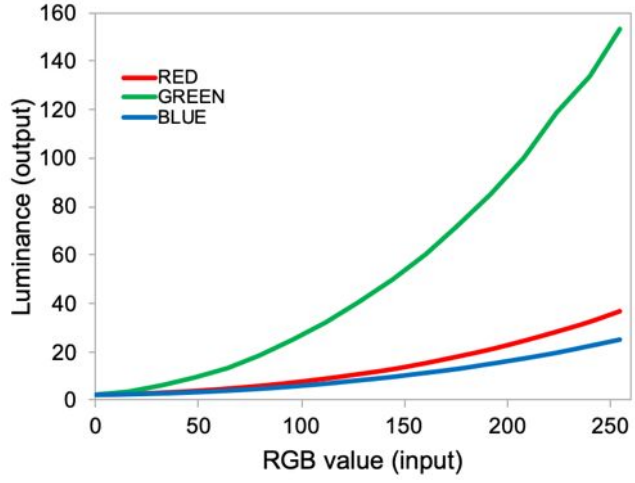

Figure 2-3 *The non-linear output of computer monitors generally follows the gamma function. This illustration shows how the screen luminance, measured with a photometer, varies with the requested RGB values.*

For many tasks, the non-linear video output is usually not a big issue, but it can easily mess up a psychophysics experiment that measures, say, perceptual

thresholds. So, we need gamma correction in some experimental scenarios to linearize the monitor output. LCD monitors typically come with a gamma of 2.2, but it is vital to measure the gamma of each monitor to get a precise gamma value you can use in your experimental task. PsychoPy has functions for deriving the gamma value of a monitor. We will introduce these functions in Chapter 9.

Vertical blanking

A CRT monitor displays an image by scanning a beam of electrons across the screen in a pattern of horizontal lines. At the end of each line, the beam returns to the start of the next line (horizontal retrace); at the end of the last line, the beam returns to the top of the screen (vertical retrace). The monitor blanks the beam to avoid displaying a retrace line during horizontal and vertical retraces. The refresh rate of a monitor describes precisely how fast it draws a full screen. A 60Hz monitor will redraw the screen 60 times a second; to draw a full screen will take 1000/60 = ~16.67 ms.

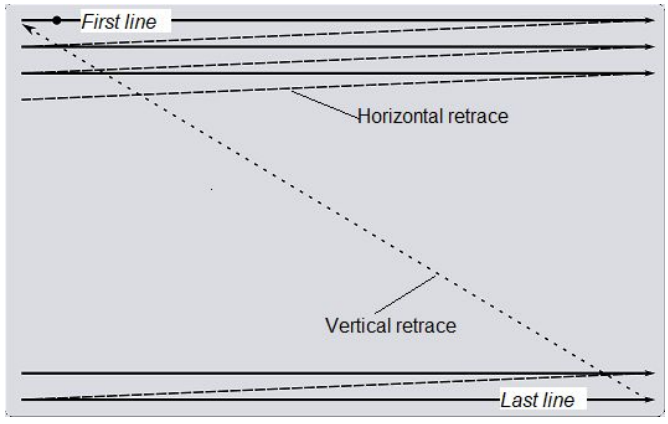

Figure 2-4 *Horizontal and vertical retraces (adapted from WikiMedia).*[5]

The implication is that the pixels of an image will not show up altogether, but instead, they appear on screen one-by-one. We can request the monitor to draw an image any time we want, but the drawing of the image will not start until the next vertical retrace. Consequently, the time at which the computer requests the

[5] Licensed to use in public domain, https://commons.wikimedia.org/wiki/File:Raster-scan.svg

drawing may not be the time at which the drawing begins on the monitor. Visual stimuli are first drawn in a memory buffer, and we need to call the *flip*() function to show the contents in the buffer on the screen. Setting the 'waitBlanking' argument to True will request the *flip*() function to block the execution of the code until a vertical blanking signal is received. So, the time returned by *flip*() is the actual start time of a new screen (and stimuli).

The key message here is always set 'waitBlanking' to *True*, and run the task in full-screen mode, which will give you better timing.

Call on flip

One useful method of a window object is *callOnFlip*(), i.e., execute something at the same time we *flip*() the window. This is especially useful in EEG studies where we need to send a TTL (e.g., through a parallel port) at the same time of stimulus onset to mark its onset in the EEG data stream. A friendly reminder here is that one should not do anything time-consuming in these function calls (e.g., saving a screenshot). The first argument to *callOnFlip*() is the function you would like to call; the other arguments are the ones as you would pass to that function. In the example script below, we define a dummy function for TTL triggering, *send_ttl(code)*, in which the 'code' argument is the TTL signal we want to send (e.g., 100). Here, we check whether the timing of the *callOnFlip*() function matches that returned by *flip*(). They should match if the function you called (e.g., send_ttl) is not time-consuming.

```
# Filename: window_callonFlip.py

from psychopy import visual, core

win = visual.Window(size=[800,600], units="pix")

# define a dummy TTL triggering function
def send_ttl(code):
    current_t = core.getTime()
    print('TTL-%d being sent at time: %d' % (code, current_t))

for ttl in range(101, 105):
    win.callOnFlip(send_ttl, ttl)
    flip_t = win.flip()
    print('Actual flipping time: %d'%flip_t)
```

```
    core.wait(1.0)

# quit PsychoPy
win.close()
core.quit()
```

Screen capture

The *visual.Window*() class has quite a few useful features. For instance, the *fps*() call will return an estimation of the frame rate since the last call to this function (or since the open of the window). The *getActualFrameRate*() call will run a test and report the frame rate of the screen. One can also capture the screen contents and save as images or videos, with the *getMovieFrame*() and *saveMovieFrames*() methods. The short script below illustrates these features. Note that taking a screenshot can be time-consuming, and I would not recommend it for timing critical tasks.

```
# Filename: screen_capture.py

from psychopy import visual, core

# create a window
win = visual.Window(size=[800,600], units="pix")

# get monitor frame rate
fps = win.getActualFrameRate()
print('Frame rate is: %d FPS'%fps)

# capture the screen
win.color=(0,0,0)
win.getMovieFrame()

# show the screen for 1.0 second
win.flip()
core.wait(1.0)

# save captured screen to a JPEG
win.saveMovieFrames("gray_window.jpg")

# quit PsychoPy
win.close()
core.quit()
```

The following message in the PsychoPy Output window reveals that the actual frame rate of my screen is 59 FPS (frame per second).

```
Frame rate is: 59 FPS
```

Visual stimuli

One of the major strengths of PsychoPy is the variety of visual stimuli supported by it, for instance, shapes, images, texts, and patterns (e.g., checkerboard). This section will briefly introduce some of the most frequently used ones.

Shapes

One can draw various shapes in PsychoPy, for instance, rectangles, circles, polygons, and of course, lines. In addition to dedicated shape constructors, i.e., *visual.Rect*(), *visual.Circle*(), *visual.Polygon*() and *visual.Line*(), PsychoPy also has an abstract class called *visual.ShapeStim*(), which is capable of drawing any shapes with arbitrary numbers of vertices. The code below shows how to draw lines and rectangles with *visual.ShapeStim*(); for lines, the "*closeShape*" argument needs to be *False*. The other shape constructors, for instance, *visual.Polygon()*, are all special cases of *visual.ShapeStim*().

```
# Filename: shapes_demo.py

from psychopy import visual, event, core

win = visual.Window(size=[800, 600], units='pix')

# line
line_vertices = [(-400,-300), (400,300)]
line = visual.ShapeStim(win, vertices=line_vertices, lineColor='white',
closeShape=False)

# rectangle
rect_vertices = [(-400,-300), (-320,-300), (-320,-240), (-400,-240)]
rect = visual.ShapeStim(win, vertices=rect_vertices, fillColor='blue',
lineWidth=0)

# draw a polygon
```

```
poly = visual.Polygon(win, edges=6, radius=100, fillColor='green')

while True:
    if rect.overlaps(poly):
        poly.fillColor='red'
    else:
        poly.fillColor='green'
    line.draw()
    poly.draw()
    rect.draw()
    win.flip()
    rect.pos += (4,3) # rectangle moving along the line
    if rect.contains((400,300)):
        break

# quite PsychoPy
win.close()
core.quit()
```

The short script above will continuously update the position of the rectangle, so it moves along the line we draw on the screen. You can, of course, dynamically change the color, size, etc. of the rectangle if needed in your task.

```
rect.pos += (4,3)
```

One useful feature of *visual.ShapeStim*() is the *contains()* method, which checks if a screen point (x, y) is inside the border of a shape. In the example script above, we use *contains()* to check if the top-right corner of the screen is in the rectangle; if so, we terminate the task.

As noted above, one can also call *visual.Rect*() etc. to draw various shapes in PsychoPy. In the example script above, we draw a hexagon at the center of the screen with *visual.Polygon*(). With this class, one can specify how many edges a polygon should have (6 edges for a hexagon, of course) and also the radius.

```
poly = visual.Polygon(win, edges=6, radius=100, fillColor='green')
```

Another useful feature of *visual.Shape*() is that it allows you to check if one shape overlaps with another with the *overlaps()* call. In the example above, we

check if the rectangle overlaps with the hexagon. If so, we change the color of the hexagon to 'red'.

```
if rect.overlaps(poly):
    poly.fillColor='red'
else:
    poly.fillColor='green'
```

GratingStim

GratingStim is one of the most frequently used stimuli in PsychoPy. It is a texture behind an optional transparency mask (filter). Both the texture and mask can be arbitrary bitmaps that repeat (cycle) in either dimension, for instance, gratings, Gabor patches (gratings with a Gaussian mask), and checkerboards. Here we will briefly introduce some of the frequently used options, of course, with examples.

```
visual.GratingStim(win, tex='sin', mask='none', units='', pos=(0.0, 0.0),
size=None, sf=None, ori=0.0, phase=(0.0, 0.0), texRes=128, rgb=None,
dkl=None, lms=None, color=(1.0, 1.0, 1.0), colorSpace='rgb', contrast=1.0,
opacity=1.0, depth=0, rgbPedestal=(0.0, 0.0, 0.0), interpolate=False,
blendmode='avg', name=None, autoLog=None, autoDraw=False, maskParams=None)
```

To use *visual.GratingStim()*, one needs to set the texture and an optional mask. The texture ('*tex*') option could be '*sin*' (sine wave), '*sqr*' (square wave), '*saw*' (sawtooth wave), '*triangle wave*', or *None* (default). The texture can also be an image with square power-of-two dimensions (e.g., 128 x 128), or a Numpy array with values ranging between -1 and 1. The mask ('*mask*') option could be set to '*circle*', '*gauss*' (2D Gaussian filter), '*raisedCos*' (raised-cosine filter), '*cross*', or *None*. [6]

Another critical parameter is spatial frequency. We should consider the screen units when setting the spatial frequency ('*sf*') option. If the screen unit is pixel ('pix), the spatial frequency has to be a fraction of 1, e.g., *sf* = 1/20, as there is no way to repeat a bitmap within a single pixel.

[6] Sebastiaan Mathot has a very good tutorial on customizing texture and mask in this blog, http://www.cogsci.nl/blog/tutorials/211-a-bit-about-patches-textures-and-masks-in-psychopy.

```
no_texture = visual.GratingStim(win, tex=None, mask=None, size=128,
pos=(200,-100))

# show the stimuli
grating.draw()
gabor.draw()
checker.draw()
numpy_texture.draw()
image_texture.draw()
no_texture.draw()
win.flip()
win.getMovieFrame()
win.saveMovieFrames('gratings.png')

# wait for 5 seconds and close the window
core.wait(5)
win.close()
core.quit()
```

The above example script set the texture to a square wave, a sine wave, a checkerboard, an image, and an 8 x 8 Numpy array with random values ranging between -1 and 1. The mask is set to None, “gauss”, “circle”, or “raisedCos”. If both the texture and mask parameters were both *None*, the result is a bright square, as the default contrast is 1 (maximum). To run this script, one needs a 128 x 128 (pixel) PNG image file named “texture.png”. Please see Figure 2-5 for resulting screen.

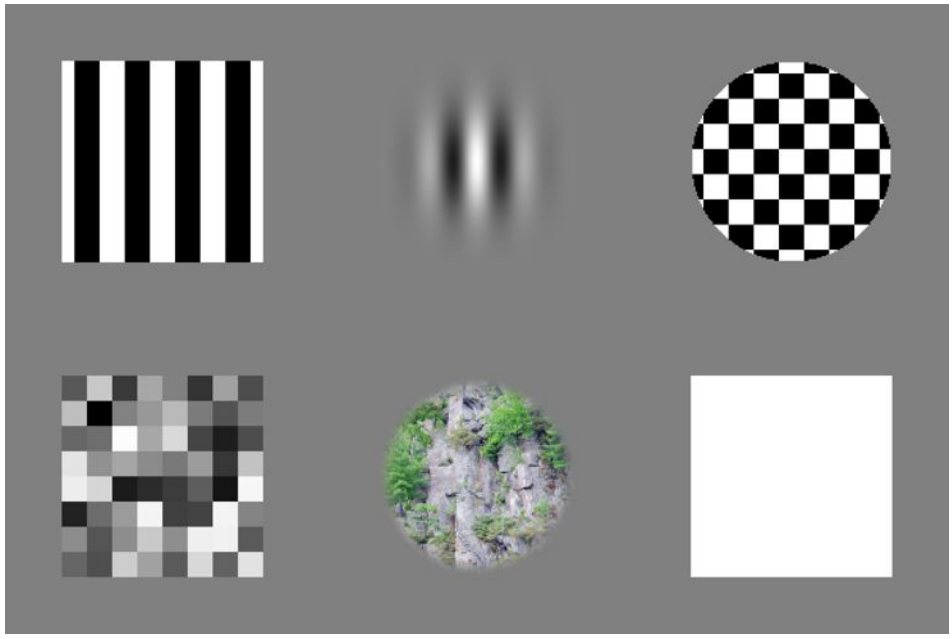

Figure 2-5 *We can use visual.GratingStim() to draw various stimuli, from a black square to a random mask. The script used to generate this image was demo_GratingStim.py (see example scripts accompanying this book).*

TextStim

Presenting texts on the screen is not as trivial as it sounds. While it is relatively simple to show text messages to the subjects, correctly formatting text for a serious reading task can be a great challenge. For instance, in an eye-tracking task, it is preferable to segment the text into interest areas to facilitate later data analysis. In PsychoPy, one can use *visual.TextStim*() or *visual.TextBox*() to present text stimuli. It is important to bear in mind that text rendering can be slow in PsychoPy, so the *draw*() call may take a bit longer time to complete, compared to other visual stimuli. So, dynamically updating the text at run time may not be a great idea in certain scenarios. The *visual.TextStim*() class has many useful features; we will highlight a few here. For an example script, please see the section on Aperture.

```
visual.TextStim(win, text='Hello World', font='', pos=(0.0, 0.0), depth=0,
rgb=None, color=(1.0, 1.0, 1.0), colorSpace='rgb', opacity=1.0,
contrast=1.0, units='', ori=0.0, height=None, antialias=True, bold=False,
```

```
italic=False, alignHoriz='center', alignVert='center', fontFiles=(),
wrapWidth=None, flipHoriz=False, flipVert=False, languageStyle='LTR',
name=None, autoLog=None)
```

Font and font files

The 'font' argument requires the name of a system font that the text should be displayed in. Exactly which font 'names' can be passed to 'font' will depend on the platform. On macOS, the names of the fonts can be found with the Font Book application. On Windows, one can list all the font names by going to the Fonts section of the Control panel. There is no easy way to retrieve the names of all available system fonts programmatically, as this is not implemented in the default graphics backend—pyglet. However, because the matplotlib library is a dependency for PsychoPy, one can use the *matplotlib.font_manager* module to retrieve the system fonts. In the code below, the *get_fontconfig_fonts*() call will return all the font files, then we use the *get_name*() method of a *FontProperties*() object to retrieve the name of the fonts.

```
from matplotlib import font_manager
f_list = font_manager.get_fontconfig_fonts()
f_names = [font_manager.FontProperties(fname=f).get_name() for f in flist]
```

Using system fonts may not be a great idea as the fonts available on macOS may not exist on Windows. It is thus preferable to include the 'fontFiles' argument, which is a list of font files (including the full path). For instance, *fontFiles = ['simHei.ttf', 'simSun.ttf']*, assuming that these two font files are stored in the same folder as the experimental script.

Right-to-left text

You can only set this parameter when initializing the *TextStim()* object. This parameter helps to correctly display texts from some languages that are written right-to-left (e.g., Hebrew). The default value for this parameter is *'LTR'* (left-to-right, case sensitive); setting to *'RTL'* will correctly display text in right-to-left languages.

Wrap width

This parameter specifies the width the text should run before wrapping.

```
text = visual.TextStim(myWin, text="Moving window example by Zhiguo"*32,
height=30,color='black', wrapWidth=760)
```

If we do not set the unit of the *TextStim*() explicitly, the unit of 'wrapWidth' will be the same as the screen.

Non-ASCII characters

One can also present non-ASCII characters (e.g., Chinese) by adding a 'u' prefix to the text string. For instance,

```
text = visual.TextStim(myWin, text=u"Example by 治国 "*32, font='STHeiti')
```

Aperture

The *visual* module of PsychoPy allows users to use an "aperture" to hide or reveal part of a window. To use the aperture function, one needs to open a window with the '*allowStencil*' option.

```
win = visual.Window(size=(800,600), units="pix", allowStencil=True)
```

Setting an aperture requires just one line of code. The example code below initializes a circular shaped aperture. The '*shape*' parameter can be '*circle*', 'triangle', or, more usefully, a list containing the vertices of a freehand shape. If needed, one can set the '*inverted*' argument to *True* to show the contents outside the aperture instead. In combination with a high-speed eye-tracker, one can easily simulate a macular degeneration condition (see Figure 2-6).

```
apt = visual.Aperture(win, size=300, shape='circle')

# specify the vertices of a freehand shaped aperture
vert = [(.1, .50), (.45, .20), (.10, -.5), (-.60, -.5), (-.5,.20)]
apt = visual.Aperture(win, size=100, shape=vert, inverted=True)
```

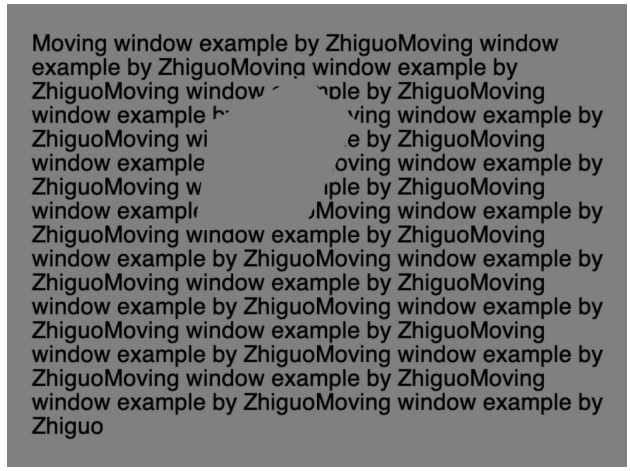

Figure 2-6 *Using aperture to simulate a macular degeneration condition.*

The short script that created the screenshot above is listed below. Note that this script uses the mouse cursor to simulate the gaze position, and the blind spot moves with the mouse.

```
# Filename: demo_aperture.py

from psychopy import visual, core, event

# create a window
win = visual.Window(size=(800,600), units="pix", fullscr=False,
                    color=[0,0,0], allowStencil=True)

# create an aperture
#apt = visual.Aperture(win, size=300, shape='circle', inverted=True)
vert = [(0.1, .50), (.45, .20), (.10,-.5), (-.60, -.5), (-.5,.20)]
apt = visual.Aperture(win, size=200, shape=vert, inverted=True)
apt.enabled = True

#create a mouse instance
mouse = event.Mouse(visible=False)

# prepare the stimuli
text = visual.TextStim(win, text="Moving window example by Zhiguo"*24,
                       height=30,color='black', wrapWidth=760)

# mouse-contingent moving window
while not event.getKeys():
```

```
    apt.pos = mouse.getPos()
    text.draw()
    win.flip()
    win.getMovieFrame()
    win.saveMovieFrames('aperture_demo.png')

# quit PsychoPy
win.close()
core.quit()
```

PsychoPy also supports other types of visual stimuli, for instance, random dots, radial stimuli (e.g., rotating wedges), rating scales, and video, etc. There is no need to go through all supported visual stimuli in detail. I would encourage users to take a look over the various example scripts that are accessible from the PsychoPy Coder menu.

Mouse and keyboard

Keyboard and mouse are the most frequently used response devices in experiments. Here we will briefly introduce some of the mouse and keyboard functions available in PsychoPy.

Mouse

The mouse is an indispensable interaction channel in tasks that require subjects to respond on a Likert scale. Some researchers have even used the mouse movement trajectories to tap into the temporal dynamics of decision making, though the timing of the mouse position samples can be unreliable. To use a mouse, one needs first to instantiate a mouse object with the *event.Mouse*() constructor. Then, one can access the various properties and methods of the *event.Mouse*() class. For instance, *getPos*() will return the current mouse position, while *setPos*() will put the mouse cursor at a given screen position. Here we present a short example script to demonstrate the most frequently used mouse functions. The task is simple: we present two options on the screen and instruct the subject to move the mouse to click one of the two options. At the end of a trial, we plot the mouse movement trajectory.

```
# Filename: mouse_demo.py
```

```
from psychopy import visual, event, core

# open a window and instantiate a mouse object
win = visual.Window(size=(800, 600), winType='pyglet', units ='pix',
colorSpace='rgb')
mouse = event.Mouse(visible=True)

# prepare the visuals
msg = visual.TextStim(win=win, text='Do you like PsychoPy?', height = 30,
pos=(0, 250))
dis = visual.TextStim(win=win, text='YES', height = 30, pos=(-200,150),
color='red')
agr = visual.TextStim(win=win, text='NO',  height = 30, pos=(200, 150),
color='green')
fix = visual.TextStim(win=win, text='+', height = 30, pos=(0, -150))
dis_box = visual.Polygon(win=win, edges=32, radius=60, pos=(-200,150),
fillColor='white')
agr_box = visual.Polygon(win=win, edges=32, radius=60, pos=(200,150),
fillColor='white')
mouse_traj = visual.ShapeStim(win=win, lineColor='black', closeShape=False,
lineWidth=5)

# use a while loop to wait for a mouse click and show the mouse trajectory
event.clearEvents()
mouse.setPos((0,-150))
traj = [mouse.getPos()] # need to call this function to update the mouse
position
while not (mouse.isPressedIn(dis_box) or mouse.isPressedIn(agr_box)):
    # if the mouse has been moved, add the new position in 'traj'
    if mouse.mouseMoved():
        traj.append(mouse.getPos())

    # put stimuli on display and draw the mouse trajectory
    msg.draw(); dis_box.draw(); agr_box.draw()
    dis.draw();agr.draw(); fix.draw()
    mouse_traj.vertices = traj # this can be slow
    mouse_traj.draw()
    win.flip()

# quit PsychoPy
win.close()
core.quit()
```

The script itself is relatively straightforward, the only thing worth mentioning here is *isPressedIn*(), which we use to check if a visual stimulus has been clicked or not. It is an excellent practice to call *event.clearEvents*() at the beginning of each trial, so cached mouse and keyboard events will not interfere with response registration. This script will show the mouse trajectory at the end of each trial (see Figure 2-7).

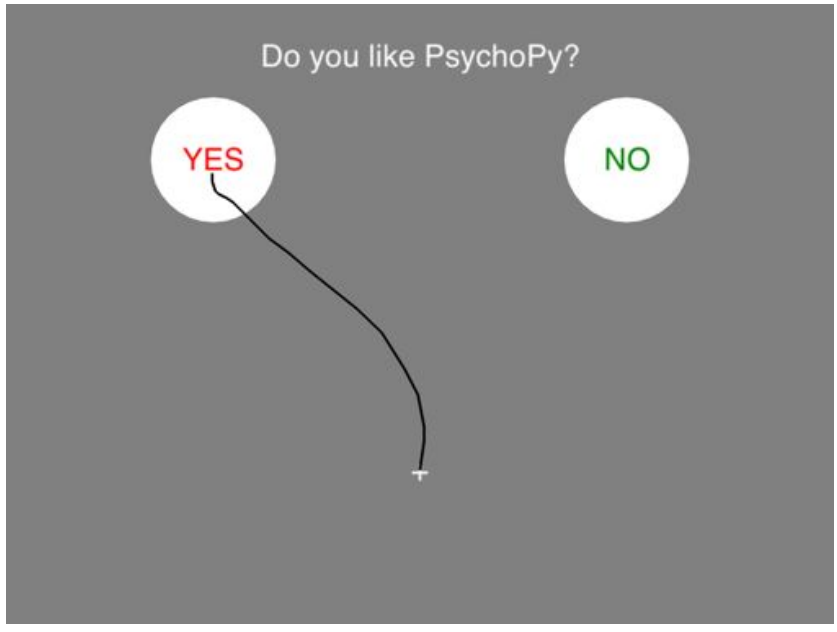

Figure 2-7 *Mouse trajectory in a decision task (see example script: mouse_demo.py).*

Keyboard

Dealing with the keyboard can be a complicated matter, as there are at least three (yes, THREE) different methods, scattered in three modules (*event*, *ioHub*, & *hardware*). The simplest solution is the *event.getKeys()* and *event.getKeys*() functions, which use the keyboard engine of Pyglet or Pygame. The timing precision of this method is not high, according to the tests done by Sol Simpson.[7] PsychoPy also comes with a unified hardware interface for keyboard and other input devices such as eye-trackers known as *ioHub*. The keyboard module of *ioHub* will give better timing precision. Yet another

[7] See the blogs posted on the LabHackers website, http://blog.labhackers.com/?p=337.

keyboard module is made available in the hardware module in PsychoPy 3. It is a wrapper of the PsychHID library of PsychToolBox, which should give you excellent timing precision that is on par with PsychToolBox. This PsychHID-based solution is what I would recommend for reaction time tasks; nevertheless, I will also introduce the other two methods in this section as they are useful for experimental tasks in which timing precision for keyboard events is not critical.

Register keyboard events with the event module

The event module offers two simple functions for keyboard events registration. The *event.waitKeys*() function halts everything while waiting for an input from the keyboard. The *event.getKeys*() function, on the other hand, is usually used to continually check if any key is pressed in a while loop. Both functions require users to specify a *keyList* argument, i.e., *keyList = ['z', 'slash']* will filter out all keys except for z and slash. The tricky thing is the names of the keys (e.g., 'enter', 'backslash') are not documented. Nevertheless, it is trivial to figure out the key names with a few lines of code. As shown in the code snippet below, we need to open a window to detect any key presses.

```
from psychopy import visual, event, core

my = visual.Window(size=(100, 100))
while True:
    key = event.waitKeys(timeStamped=True)
    print(key)
```

As shown in the shell output below, the *event.waitKeys*() will return a list of pressed keys. Because we set the *timeStamped* argument to *True*, all key presses have timestamps. We should always enable this argument, as the screen *flip*() call may lead to timing errors up to a screen refresh.

```
[['a', 5.818389608990401]]
[['b', 6.602048815984745]]
[['apostrophe', 13.980140805011615]]
[['slash', 15.94004777900409]]
[['backslash', 18.18008749600267]]
[['return', 19.80443760601338]]
```

Register keyboard events with the ioHub module

The rationale behind *ioHub* is that hardware events (keyboard and mouse, etc.) should be monitored in a thread that is parallel to experimental control (e.g., the presentation of stimuli), so the timing of the hardware events will not be affected by factors like screen refreshes. The devices currently supported by *ioHub* are limited to keyboard, mouse, and eye-tracker. To use *ioHub* one needs first to launch an *ioHub* server, which will be running in a separate thread in the background. Then, we can refer to the keyboard and call functions like *getEvents*(), *waitForKeys*(), *waitForPresses*(), and *waitForReleases*().

```
from psychopy import core, iohub

io = launchHubServer() # start ioHub service
kb = io.devices.keyboard # refer to the keyboard device
s_time = core.getTime()
while core.getTime()-s_time < 5.0:
    for ev in kb.getEvents():
        print(ev)

io.quit() # Stop the ioHub Server
```

Register keyboard events with PsychHID

Registering keyboard events with the Psychtoolbox PsychHID engine is a new feature of PsychoPy. This function is available on 64-bits Python 3 installations only. On 32 bit installations and Python 2, it reverts to the *event.getKeys*() call.

```
# Filename: keyboard_PsychHID.py

from psychopy.hardware import keyboard
from psychopy import core, visual

win = visual.Window((200,200))

# create a keyboard device
kb = keyboard.Keyboard()

def waitKey():
    ''' a function to detect a single key press'''
    got_key = False
    while not got_key:
```

```
        keys = kb.getKeys()
        if keys:
            for key in keys:
                print(key.name, key.duration, key.rt, key.tDown)
            got_key = True

for i in range(10):
    win.color = (i%2*1.0, -1, -1)
    win.flip()
    kb.clock.reset() # reset the clock
    waitKey()
```

The above script illustrates the basic usage of this new module. One needs first to initialize a keyboard object, which can take the following arguments.

```
keyboard.Keyboard(device=-1, bufferSize=10000, waitForStart=False,
clock=None)
```

The device argument only applies to Linux and macOS, allows users to select from multiple keyboard devices, and then call *start()* and *stop()* to switch input from different keyboard devices. When set to -1, PsychoPy will use the default keyboard. The *bufferSize* argument specifies how many key presses to store in the buffer before it starts to drop the earliest keys. One can also provide a *clock* argument to specify which *core.Clock*() object to use,[8] without which PsychoPy will use the global clock. The *waitForStart* argument is usually set to *False*, as it is generally unnecessary to manually start/stop polling the keyboard.

To poll the key events one needs to call *getKeys*(), which will return a list of Keypress object, from which the key name, the duration (from keypress to key release), the response time (from the most recent clock reset), and the timestamp of keypress can be retrieved.

Trial control

An experimental task usually involves repeated measurement of behaviors, and each measurement is known as a “trial”. A trial typically consists of multiple

[8] PsychoPy allows users to instantiate multiple clocks to keep track of time for different purposes, e.g., a global clock to keep track of the amount of time elapsed since task onset and another clock to calculate response time on each trial.

events, for instance, presenting a fixation cross, showing an image, and then waiting for a participant response. How to present the trials in a randomized order? Which trials are grouped in blocks? Should an error trial be recycled and presented to the subject again at a later time? All these questions need to be dealt with when considering the control of experimental trials. PsychoPy an experiment Handlers (e.g., a staircase handler) to control the trial presentation. Here, I will present a much simpler way of controlling trial presentations based on Python lists.

Before diving in, I would like to emphasize that putting trials in a big *for*-loop, which contains another *for* loop, is a terrible idea (I have seen too many scripts like this in Matlab). It makes the code difficult to read, even to the author himself/herself. It is usually preferable to define a "trial" function to handle the sequence of events of a single trial. This trial function, of course, will need to accept task parameters to change the behavior of each trial. Then, trial control becomes a simple matter of specifying the parameters of all trials in an iterable structure (e.g., a list) or a spreadsheet that we read during testing. The following pseudo-code illustrates this simple idea.

```
# define a trial function
def run_trial(par1, par2, par3):
    '''present the trial events based on the parameters'''

    do_something here
    return trial_result

# specify the parameters of all unique trials in a list
trials = [[t1_par1, t1_par2, t1_par3],
            [t2_par1, t2_par2, t2_par3],
            ...
            [tn_par1, tn_par2, tn_par3]]

# iterate over all the trials
for trial_pars in trials:
    par1, par2, par3 = trial_pars
    run_trial(par1, par2, par3)
```

Assume that we would like to manipulate three parameters in each experimental trial (par1, par2, par3). For a single trial, a combination of the values for these three parameters are stored in a list or a tuple, e.g., [t1_par1,

t1_par2, t1_par3] for trial 1. We then put these lists into another list, as we did in the pseudo-code above. Then, we should be able to loop through the list and pass the trial parameters to the trial function.

If we need each of the unique trials to repeat 24 times, multiply the unique trial list by 24, e.g., *trial2Test = trials[:] *24*. Then, call the *random.shuffle*() function from the *random* module to randomize the trials, e.g., *random.shuffle(trial2Test)*. Then, run a for-loop to iterate over all trials.

The *run_trial* function in the above pseudo-code does not need to return any value, but a return value can be useful if you need to recycle error trials. For instance, the script below allows the participant to press the right arrow key to pass a trial and to push the left arrow key to recycle a trial. If you press the right arrow key, the *run_trial()* function returns True; if you press the left arrow key , the *run_trial()* function returns False. In a while loop, we randomly select a trial from the *trial_list* and test it. If the return value *should_recycle == False*, we remove the trial from the *trial_list*; otherwise, the trial will remain in the *trial_list*, until the while loop ends, i.e., after all of the trials have been completed.

```
# Filename: simon_effect.py

import random
from psychopy import visual, event, core

myWin = visual.Window(size=(600,400), units ='pix')

def run_trial(trial_id):
    """a simple function to run a single trial"""

    # show some info on the screen
    msg1 = visual.TextStim(myWin, text='This is
Trial----{}'.format(trial_id), pos=(0,100))
    msg2 = visual.TextStim(myWin, text='RIGHT--> Next trial; LEFT--> Recycle
current trial')
    msg1.draw(); msg2.draw()
    myWin.flip()

    # wait for a response
    key = event.waitKeys(keyList=['left', 'right'])
```

```
    # clear the window
    myWin.clearBuffer()
    myWin.flip()
    core.wait(0.5)

    if 'right' in key:
        recycle = False
    if 'left' in key:
        recycle = True

    return recycle

trial_list = ['t1', 't2', 't3', 't4', 't5']

# recycle trials with a while loop
while len(trial_list) > 0:
    # randomly select a trial from the trial_list
trial_to_test = random.choice(trial_list)

    # run a single trial, the return value of run_trial could be True or
False
should_recycle = run_trial(trial_to_test)

    # remove the trial we just tested from the list if the return value is
False
    # (i.e., no need to recycle)
    if should_recycle == False:
        trial_list.remove(trial_to_test)

    # show what trials are left in the trial list
    print('Trials left in the list: {}'.format(trial_list))

# quit PsychoPy
core.quit()
```

Figure 2-8 shows a screenshot of the above script in action. The trial currently being tested is "t1". If we keep pressing the left arrow key, "t1" will remain in the *trial_list*.

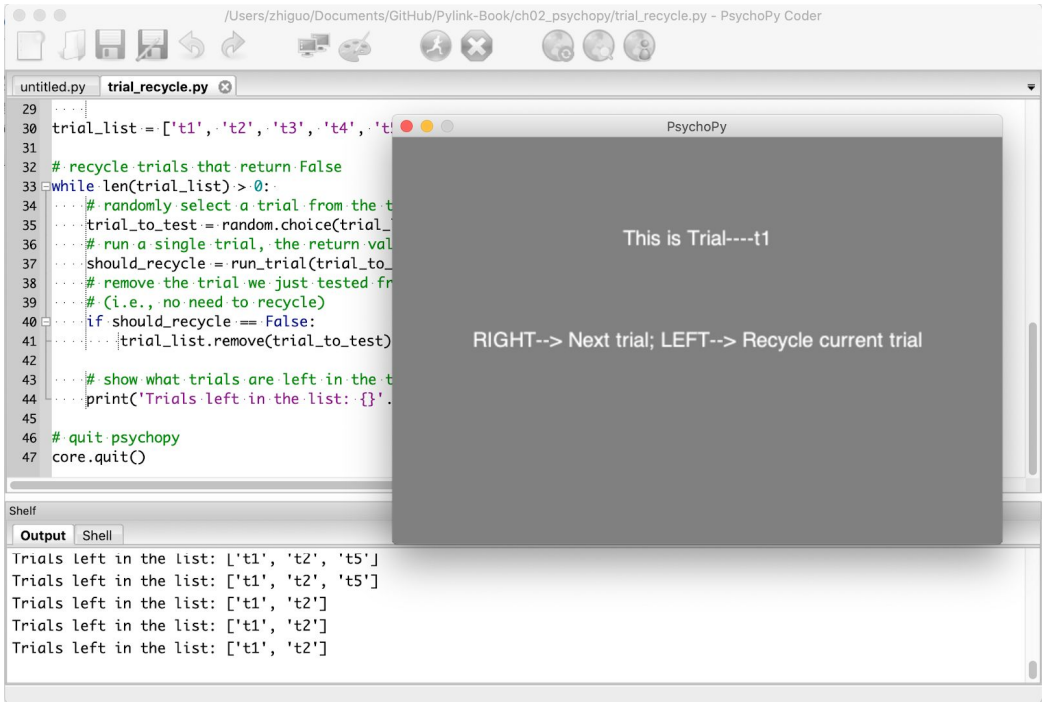

Figure 2-8 *An example script for trial recycling. In the while-loop, we randomly select a trial from a trial list; it is then removed from the list if the return value (should_recyle) is False.*

A real example: Simon effect

The previous sections have covered all the basics one needs to build an experimental task. This section will give a short example script demonstrating the famous Simon effect, i.e., a stimulus-response compatibility effect. In this task, the participant responds to the color of a disk that could appear on the left or right side of the screen. The color of the disk dictates which button (hand) – left or right—is the correct response, e.g., left button for the red disk and right button for the green disk. The location of the disk is irrelevant. If the disk appeared on the left side of the screen and the subject responded with the left hand, that will give us a *congruent* trial. However, if the response requires the use of the right hand, that will provide us with an *incongruent* trial. The robust finding is a response cost on incongruent trials, i.e., prolonged response time if the target requires a response from a hand on the opposite side.

This is, of course, an over-simplified introduction of the Simon effect; interested readers, please refer to the review by Lu and Proctor (1995).

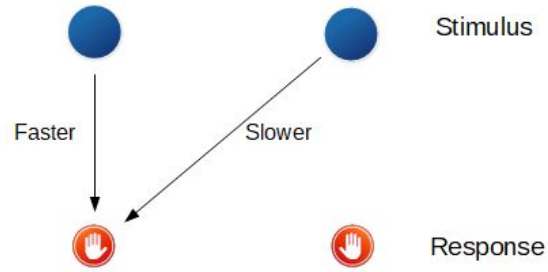

Figure 2-9 *An illustration of the Simon task; adapted from Lu and Proctor (1995). The subject responds to the color of the disk; the response time is longer when the stimulus appears on the opposite side (relative to the responding hand).*

The script for this task is listed below. It is quite impressive that less than 80 lines of code (including empty lines and comments) are sufficient to implement the Simon task.

```
# Filename: simon_effect.py

import random
from psychopy import visual, core, event, gui

# open a window and create stimuli
win = visual.Window((1024, 768), units='pix', fullscr=False, color='black')
text_msg = visual.TextStim(win, text='message')
tar_stim = visual.GratingStim(win, tex='None', mask='circle', size=60.0)

# possible target position
tar_pos = {'left': (-200, 0), 'right':(200, 0)}

# trial list
trials = [['left', 'red', 'z', 'congruent'],
          ['left', 'blue', 'slash', 'incongruent'],
          ['right', 'red', 'z', 'incongruent'],
          ['right', 'blue', 'slash', 'congruent']]

def run_trial(trial_pars, data_file, participant):
    """ Run a single trial.
```

```
    trial_pars -- target position, color and correct key, e.g., ['left',
'red', 'z', 'cong']
    data_file -- an file to save trial data
    participant -- information about the participant in a dictionary,
{'id':1, 'name':zw}"""

    pos, color, cor_key, congruency = trial_pars # upacking the parameter
list
    tar_stim.pos = tar_pos[pos] # set target postion
    tar_stim.color = color # set target color

    # present the fixation for 750 ms
    text_msg.text = '+'
    text_msg.draw()
    win.flip()
    core.wait(0.750)

    # present the target and wait for a key response
    tar_stim.draw()
    win.flip()
    t_tar_onset = core.getTime()
    tar_resp = event.waitKeys(1500, ['z', 'slash'],timeStamped=True)

    # write data to file
    trial_data = list(participant.values()) + trial_pars +[t_tar_onset] +
list(tar_resp[0])
    trial_data = map(str, trial_data) # convert to string
    data_file.write(','.join(trial_data) + '\n')

    # clear the screen and set an ITI of 500 ms
    win.color = 'black'
    win.flip()
    core.wait(0.500)

# -- real experiment starts here --
# get participant info from a dialog
participant = {'Participant ID': 0, 'Participant Initials': 'zw'}
dlg = gui.DlgFromDict(participant, title='Enter participant info here')

# open a data file with write permission
d_file = open(participant['Participant Initials']+'.csv', 'w')

# show the task instructions
```

```
text_msg.text = 'Press Z to RED\nPress / to BLUE\n\nPress <SPACE> to start'
text_msg.draw()
win.flip()
event.waitKeys(keyList=['space'])

# randomly shuffle the trial list and iterate over all of them
random.seed = 1000
test_trials = trials[:]*2
random.shuffle(test_trials)
for pars in test_trials:
    run_trial(pars, d_file, participant)

# close data file
d_file.close()

# exit
core.quit()
```

The script begins with the importing of necessary libraries and modules, of course. Note that, in addition to the frequently used PsychoPy modules (visual, event, and core), we also import the *random* module to handle trial randomization.

```
import random
```

As usual, we need to open a window to present the stimuli; then, we instantiate the target stimulus in memory with *visual.GratingStim*(). Note that setting the "*tex*" parameter to *None* and the "*mask*" parameter to "*circle*" will give us a *visual.GratingStim*() with maximum contrast (i.e., a filled disk).

```
# open a window and create stimuli
win = visual.Window((1024, 768), units='pix', fullscr=False, color='black')
# target disk
tar_stim = visual.GratingStim(win, tex='None', mask='circle', size=60.0)
```

For this simple task, the parameters of a trial include target position, target color, correct response key, and congruency. Assume that the target appears on the left side of the screen; if the required response key is 'z', we will have a congruent trial; if the required response key is '/' (slash), we will have an incongruent trial. The participant is required to press 'z' to red targets and '/' to

blue targets. Consider all possible values for each of these parameters; we can construct a list of unique trials.

```
trials = [['left', 'red', 'z', 'congruent'],
          ['left', 'blue', 'slash', 'incongruent'],
          ['right', 'red', 'z', 'incongruent'],
          ['right', 'blue', 'slash', 'congruent']]
```

Here the parameters of each unique trial are stored in a list, e.g., *['left', 'red', 'z', 'congruent']*; the parameters of all possible trials are a list of lists, i.e., the value stored in the variable *trials*. At the beginning of each trial, we take a list from *trials*, unpack the parameters (using indexing) and pass them to the run_trials function to control the stimulus position and color, etc. for that trial. Note that, to make the result file more accessible, we specified 'left' and 'right' for the target position, instead of the actual screen pixel coordinates in the parameter list.

```
pos, color, cor_key, congruency = trial_pars # upacking the parameter list
tar_stim.pos = tar_pos[pos]
```

We store the possible target positions in a dictionary called *tar_pos*, so, for example, *tar_pos['left']* will give us (-200, 0).

```
tar_pos = {'left': (-200, 0), 'right':(200, 0)}
```

The most crucial bit of this script is, of course, the *run_trial()* function. This function takes three arguments, i.e., a list that stores the trial parameters (*pars*), an open file to log the data collected for each trial (*data_file*), and a dictionary storing participant information (*subje_info*).

```
def run_trial(trial_pars, data_file, participant):
    """ Run a single trial.

    trial_pars -- target position, color and correct key, e.g., ['left',
'red', 'z', 'cong']
    data_file -- an file to save trial data
    participant -- information about the participant in a dictionary,
{'id':1, 'name':zw}"""
```

We first unpack the trial parameters (*trial_pars*) and change the color and position of the target disk. Then, we present the fixation cross and the target and wait for the subject to issue a response.

```
pos, color, cor_key, congruency = trial_pars # upacking the parameter list
tar_stim.pos = tar_pos[pos] # set target postion, e.g., tar_pos['left']
tar_stim.color = color # set target color
```

Here, we use a simple dialog to collect the participant information. Simple dialog windows are available through the *gui* module of PsychoPy. In the example script, we use a dictionary to construct the dialog and to store the input by the experimenter.

```
participant = {'Participant ID': 0, 'Participant Initials': 'zw'}
dlg = gui.DlgFromDict(participant, title='Enter participant info here')
```

At the end of a trial, we concatenate the participant information, the trial parameters, and the response data into a long list. We first use the *map* function to convert all items in the list into strings (by applying the *str* function). Then, we use the *join()* method to connect the items in the list with commas and save the resulting string to file.

```
# write data to file
trial_data = list(participant.values()) + trial_pars +[t_tar_onset] +
list(tar_resp[0])
trial_data = map(str, trial_data) # convert to string
data_file.write(','.join(trial_data) + '\n')
```

Although this data logging method may appear low-tech, it is highly reliable. Because we write data to a plain text file at the end of each trial, even if a task crashes in the middle of a testing session, no data collected so far will be lost. I have never lost any data with this simple data logging method. The resulting .csv file can be loaded into Excel or R for further examination (see Figure 2-10).

J9

	A	B	C	D	E	F	G	H	I
1	2	zw	right	red	z	incongruent	11.071	z	11.771
2	2	zw	right	blue	slash	congruent	13.069	slash	13.627
3	2	zw	left	blue	slash	incongruent	14.919	slash	15.555
4	2	zw	left	blue	slash	incongruent	16.853	slash	17.41
5	2	zw	right	red	z	incongruent	18.708	z	19.283
6	2	zw	left	red	z	congruent	20.576	slash	21.203
7	2	zw	right	blue	slash	congruent	22.493	slash	23.307
8	2	zw	left	red	z	congruent	24.596	z	25.043

Figure 2-10 *The output from a testing session of 8 trials.*

At the beginning of the testing session, we first collect participant information with the pop-up dialog, then show the task instructions. Note that, before we shuffle the trial list, we set a random seed. Setting a fixed random seed may not be preferable in all tasks, but it is crucial to bear in mind that computers can only do pseudo randomization. If you use the same random seed, the computer will always give the same sequence of random numbers. The *random.shuffle()* function randomizes the list of trials; the *for*-loop helps to run the task through.

```
random.seed = 1000
test_trials = trials[:]*2
random.shuffle(test_trials)
for pars in test_trials:
    run_trial(pars, d_file, participant)
```

The features of PsychoPy covered in this chapter should be sufficient for most experimental tasks. The example code that comes with PsychoPy covers almost all aspects of PsychoPy and is a perfect place to get started or find out more about PsychoPy and its features. The online user manual of PsychoPy is also a reference source that I highly recommend.