

Programming for Psychology in Python

Vision Science

0. **Introduction**
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Introduction

Welcome to this set of lessons on using Python in vision science. The aim of these lessons is to provide you with a set of skills that you can use to develop your own visual stimuli and control your experiments.

What content will we be covering?

In this series of lessons, we will be looking at:

1. How we can use the [psychopy](#) package to implement vision science experiments.
2. The concept of a drawing window.
3. How we can draw sinusoidal gratings with various parameters.
4. How we can draw shapes and lines.
5. How we can draw images from disk.
6. How we can create stimuli from small dots.
7. Controlling the temporal dynamics of stimuli and experiments.
8. Collecting responses from participants.
9. Providing input into a program when it begins.
10. Saving data to disk.
11. A framework for assembling a vision science experiment, with an example experiment.
12. An additional example of the implementation of a vision science experiment.

How are these lessons organised?

Each lesson is focused around a key concept in the fundamentals of Python programming for vision science. The recommended approach is to follow along with the content while writing and executing the associated code.

You will also find that each lesson has an associated screencast. In each screencast, I narrate the process of coding and executing scripts related to the concept of the lesson. The purpose of these screencasts are to give a practical demonstration of the concepts of the lesson. They will typically cover much the same content as the written material. It is recommended that you both view the screencast and read through the written material.

You can also view the written material of all the lessons as a [combined PDF](#).

The lessons also contain the occasional *tip*, which are small concepts that give additional suggestions on the relevant content (sometimes highlighting very important concepts). An instructive example is:

Tip: It is best to watch the screencasts in HD resolution—otherwise the text is not very legible. If you don't have a fast enough network connection, you can copy the video (mp4) files from the course directory and view those rather than streaming.

Finally, the lessons also include exercises for you to try.

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. **Getting started**
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Getting started

Objectives

- Be able to use psychopy to open a window and draw text to the screen.
- Understand how scripts are executed and how psychopy can be used to control the flow of execution.

Screencast

In this lesson, we are going to introduce the key components of how we use psychopy to implement and control vision science experiments. We will go into these components in more detail in future lessons, but here we will take a quick

look at their basic qualities.

The first step is to make the additional functionality that is provided by the psychopy package available to our code. We do this by using the `import` statement, as usual. The psychopy package is broken up into a number of subpackages; initially, we will just be interested in the `visual` subpackage.

```
import psychopy.visual
```

A critical component of a vision science experiment is the drawing window. This is a representation of the screen to which we can draw stimuli and control the environment. We will be looking more into windows in the next lesson, but here let's define a window that is 400 pixels high and 400 pixels wide and is white in colour. We do that by:

```
import psychopy.visual

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)
```

As you can see, we specify the `size` via a two-item list, which sets the horizontal and vertical extent, respectively. We then tell psychopy that we have specified such values in units of pixels. We then set the argument `fullscr` to the boolean value `False`; if this was `True` (the default), the `size` parameter would be overridden and the window would occupy the full screen. This is desirable when actually running experiments, but while we are learning it is useful to just have it in a small window. Finally, we set its colour (note the American spelling of colour) to white. Colour in psychopy can be specified in a few different ways, but we will use the RGB format. Here, each item in the list specifies how much the red, green, and blue components (respectively) of the display should be driven, from -1 to +1.

Tip: You will also notice in the above some slightly different formatting. Many of the psychopy functions take quite a few arguments, and it can get unwieldy to specify them all on one long line. Hence, a useful strategy to improve readability is to specify the arguments on different lines. Note however that the arguments are indented one level "under" the opening and closing parentheses. This tells Python that the arguments "belong" to this function.

Now that we have created our window, let's create a basic stimulus—some text. We can do that using psychopy's `TextStim` data type. Most of its default arguments are fine for our purposes; we will just tell it about our window and what text we would like it to display:

```
import psychopy.visual

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

text = psychopy.visual.TextStim(win=win, text="Hello, world!")
```

Now we have made our text data type, we prepare to show it by using its `draw` function. To actually display what we've drawn in the window, we then have to "flip" the window (more on this next lesson). Finally, we close the window.

```
import psychopy.visual

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

text = psychopy.visual.TextStim(win=win, text="Hello, world!")

text.draw()

win.flip()

win.close()
```

If we run the above code, however, it is difficult to see the results of our handiwork. This is because the Python statements execute sequentially, and once we have drawn our stimulus the program finishes.

Tip: You may see a message in the output window with a warning about a monitor specification. This refers to the (quite useful) functionality in psychopy for managing the physical characteristics of monitors. It can be safely ignored, in this context.

What we need is a way to interrupt our program's flow. We can use functionality in psychopy's `event` package to achieve this. Specifically, we will use a function called `waitKeys`, which will halt the execution of our program until you press a key on the keyboard.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

text = psychopy.visual.TextStim(win=win, text="Hello, world!")

text.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```

If we run the above code, we can see that the window stays open while waiting for our keypress. However, we can't see any text. This is because the default colour of text is white—so we can't see it on our white window background! We can fix this by giving a `color` argument to `TextStim`.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

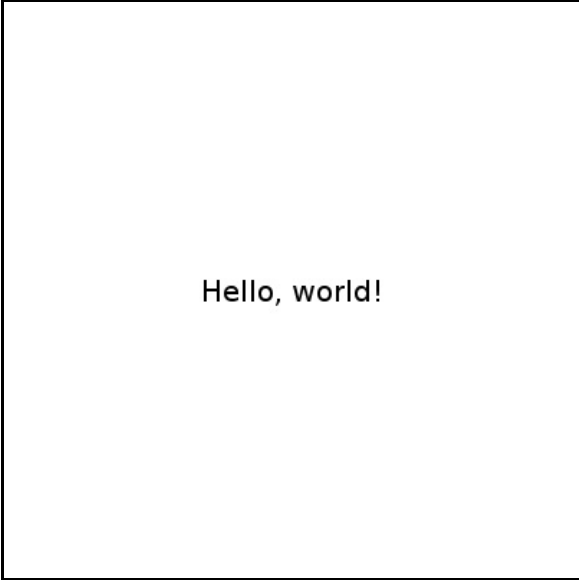
text = psychopy.visual.TextStim(
    win=win,
    text="Hello, world!",
    color=[-1, -1, -1]
)

text.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



Hello, world!

Exercises

1. Draw the text to the screen in the above example in a mid-green colour.
2. What does the `flipHoriz` argument in a `TextStim` do? Can you think of a situation where this might be useful?
3. A option for the `units` argument to many psychopy functions is "deg", which allows specification in degrees of visual angle (angular subtense). This is a very common specification in vision science, but what extra pieces of information does it require? Using this in psychopy requires creating a `Monitor`.

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. **Drawing to a window**
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Drawing to a window

Objectives

- Understand the process of drawing stimuli to the window.
- Be aware of the implications of window "flipping".

Screencast

In this short lesson, we are going to cover a simple but important concept—the process by which we can draw and present visual stimuli.

Most, if not all, of the psychopy stimulus types that we will be using are drawn to the screen using the same procedure. They each have a `.draw()` function—this tells psychopy to take the information that has been provided about the current stimulus and render it to the window. However, a very important point is that such a rendering is not immediately visible in the window.

To illustrate this, we can take our example from the previous lesson but remove the mysterious `win.flip()` command (note that the `text.draw()` command remains):

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

text = psychopy.visual.TextStim(
    win=win,
    text="Hello, world!",
    color=[-1, -1, -1]
)

text.draw()

#win.flip()

psychopy.event.waitKeys()

win.close()
```

Tip: Recall that everything to the right of a `#` character in a line of Python code is treated as a "comment", and is not interpreted by Python.

You can see that even though we have "drawn" the stimulus to the window, it does not appear. To make it appear, we have to reinstate the `win.flip()` command. So what does this command do and why does it make what we have drawn visible? A useful way to think about it is as if the window has two layers, one in front of the other. When we execute the `draw` command, we are "painting" the second layer of the window which, because it is behind the first layer, we cannot see. Once we have done all the drawing we want to do, and we are ready to see the results, we "flip" the layers around; what was at the back is now at the front, and hence visible.

Importantly, in this process of "flipping" the window, we wipe what was previously drawn to the front window. That means that each time we flip the window, we need to draw something again in order to see it. For example, we can add in another section where we flip the window and wait for another keypress, but this time without drawing our text.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

text = psychopy.visual.TextStim(
    win=win,
    text="Hello, world!",
    color=[-1, -1, -1]
)

text.draw()

win.flip()

psychopy.event.waitKeys()

win.flip()

psychopy.event.waitKeys()
```



```
win.close()
```

As you can see, the second time around does not show our text message—because we had not drawn anything to the window after we flipped it.

A final point about drawing stimuli is that the order of drawing matters, with more recent `draw` commands having to potential to overwrite what was drawn by previous commands. For example, we could draw the text in both black and green.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

text = psychopy.visual.TextStim(
    win=win,
    text="Hello, world!",
    color=[-1, -1, -1]
)

# text is black
text.draw()

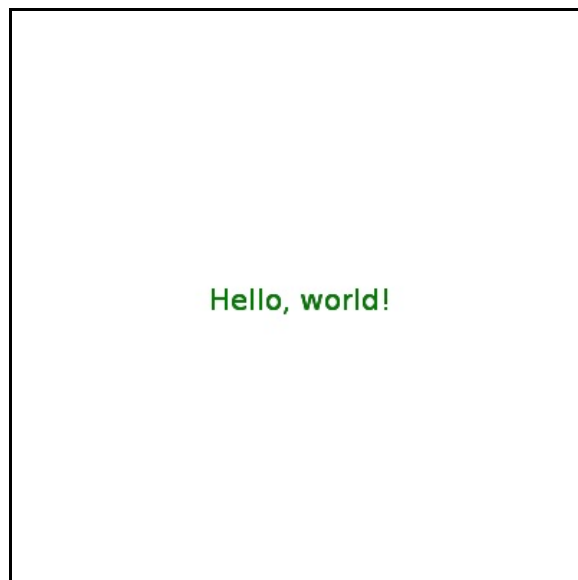
# change text to green
text.color = [-1, 0, -1]

# draw text again
text.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



As you can see, there is no sign of our black text—it has been overwritten by the green. We can see this even more clearly if we move the green text horizontally and vertically by a small amount, creating a neat shadow effect:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)
```

```
text = psychopy.visual.TextStim(  
    win=win,  
    text="Hello, world!",  
    color=[-1, -1, -1]  
)  
  
# text is black  
text.draw()  
  
# change text to green  
text.color = [-1, 0, -1]  
  
# shift the green text one pixel to the right and one pixel up  
text.pos = [1, 1]  
  
# draw text again  
text.draw()  
  
win.flip()  
  
psychopy.event.waitKeys()  
  
win.close()
```

Tip: Position values are specified in psychopy via a two-item list of coordinates (horizontal and vertical). The pair [0, 0] is at the centre of the screen. Negative horizontal coordinates are offset to the left of centre and positive to the right of centre. Negative vertical coordinates are offset below centre and positive above centre.



Exercises

1. Amend the example above to generate a stimulus like shown below:



2. Use your knowledge of text drawing, positioning, and colouring, in addition to loops, to generate a stimulus like shown below:



[Back to top](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. **Drawing—gratings**
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Drawing—gratings

Objectives

- Be able to generate and display grating stimuli.
- Understand and manipulate key grating parameters (phase, spatial frequency, orientation, contrast).
- Be aware of different ways to "mask" stimuli such as gratings.

Screencast

Gratings are a very important type of stimulus in vision research. Fundamentally, they are created from oscillations in

luminance over space (typically sinusoidal). In psychopy, we can create gratings using `GratingStim`.

Let's take a look at the grating that is created when we use mostly default parameters:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

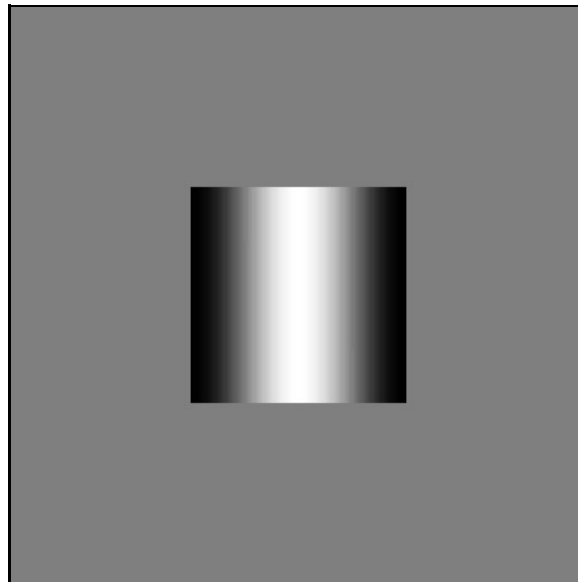
grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
    size=[150, 150]
)

grating.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



As you can see, we have created a square stimulus where the luminance intensity goes from black at the left edge, white in the middle, and back to black at the right edge. Overall, this change in luminance intensity follows a sine wave profile.

Now, we will go through some of the key arguments that are used to change the properties of grating stimuli.

Phase

This parameter controls the relative positioning of the light and dark regions of the grating. In psychopy, this parameter is specified as a value between 0 and 1. For example, the code below shows four gratings that change in phase between 0 (top) and 0.5 (bottom). Notice how the position of the "light bar" moves horizontally with the change in phase; when the phase is 0.5, the light and dark regions have swapped compared with when the phase was 0.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
```

```

    size=[80, 80]
)

grating_vpos = [150, 50, -50, -150]
grating_phase = [0.0, 0.16, 0.33, 0.5]

for i_phase in range(4):

    grating.phase = grating_phase[i_phase]

    grating.pos = [0, grating_vpos[i_phase]]

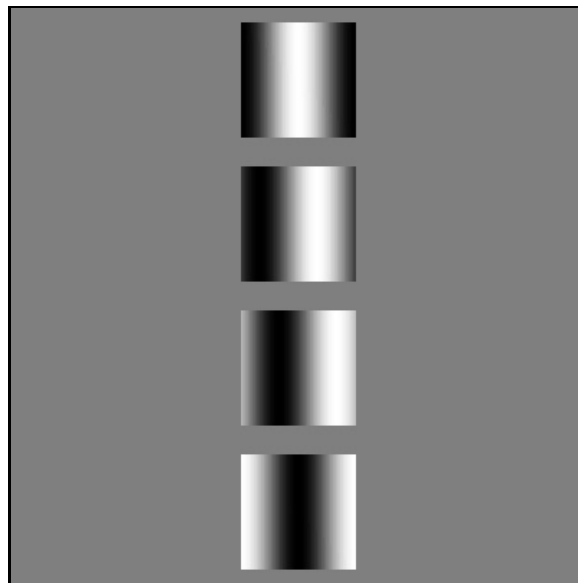
    grating.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

```



Spatial frequency

This parameter specifies the number of cycles (the number of oscillations) over some unit of distance. In psychopy (when the units are `pix`, as we typically use), this is in cycles per pixel.

For example, if we have a grating that is 150 pixels in size and we want it to have 5 cycles, the spatial frequency would be $5 / 150 = 0.033$:

```

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
    size=[150, 150]
)

grating.sf = 5.0 / 150.0

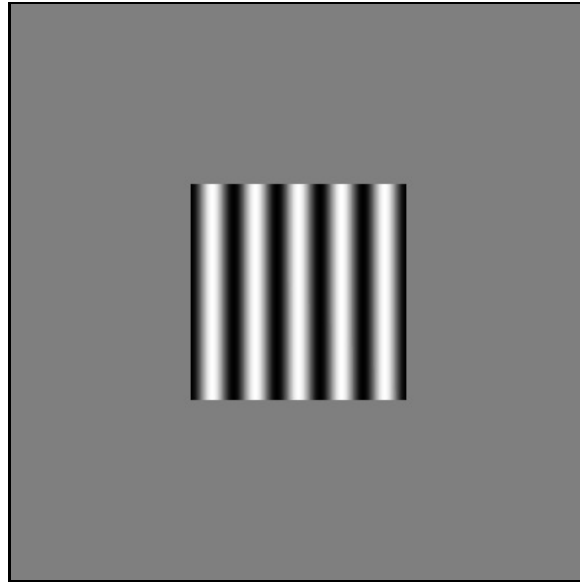
grating.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

```



Orientation

This parameter controls the 'tilt' of the grating. In psychopy, we specify this in units of degrees, where 0 is vertical and increasing angles are increasing clockwise. For example:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
    size=[80, 80]
)

grating.sf = 5.0 / 80.0

orientations = [0.0, 45.0, 90.0, 135.0]
grating_hpos = [-150, -50, 50, 150]

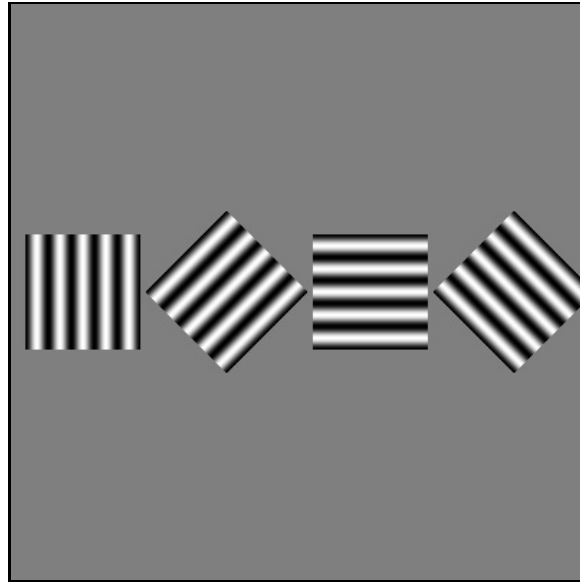
for i_grating in range(4):
    grating.ori = orientations[i_grating]
    grating.pos = [grating_hpos[i_grating], 0]

    grating.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



Masking

In the previous example, changing the orientation of the grating also changed the shape of its region. For this reason, amongst others, we typically present the grating with a "mask", which varies the visibility of the grating within its region. A typical mask is a Gaussian, which we can set by using `mask = "gauss"`, as below:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
    size=[80, 80]
)

grating.sf = 5.0 / 80.0

grating.mask = "gauss"

orientations = [0.0, 45.0, 90.0, 135.0]
grating_hpos = [-150, -50, 50, 150]

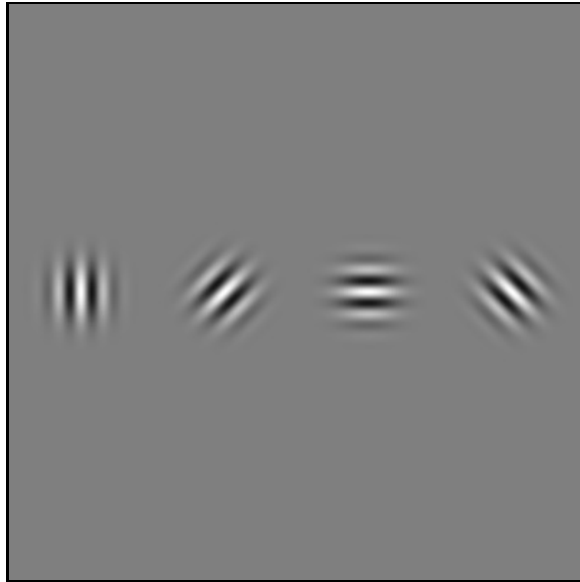
for i_grating in range(4):
    grating.ori = orientations[i_grating]
    grating.pos = [grating_hpos[i_grating], 0]

    grating.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```

Another common mask is a circle, which we can set by using `mask = "circle"`, as below:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
    size=[80, 80]
)

grating.sf = 5.0 / 80.0

grating.mask = "circle"

orientations = [0.0, 45.0, 90.0, 135.0]
grating_hpos = [-150, -50, 50, 150]

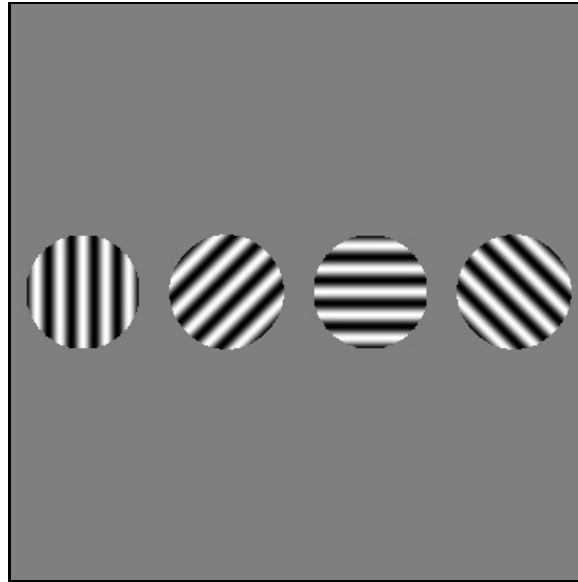
for i_grating in range(4):
    grating.ori = orientations[i_grating]
    grating.pos = [grating_hpos[i_grating], 0]

    grating.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



Contrast

The final parameter we will consider is the grating contrast. This value ranges between 0 and 1, and determines the magnitude of the difference between the intensity of the highest and lowest luminance areas in the grating (formally, it is the difference between the maximum and minimum luminances divided by the sum of the maximum and minimum luminances). When the contrast is 0, the grating is invisible, whereas when it is 1 it is at full contrast.

In the example below, we change the contrast of the gratings from low (0.1) to high (0.8):

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
    size=[80, 80]
)

grating.sf = 5.0 / 80.0

grating.mask = "circle"

contrasts = [0.1, 0.2, 0.4, 0.8]
grating_hpos = [-150, -50, 50, 150]

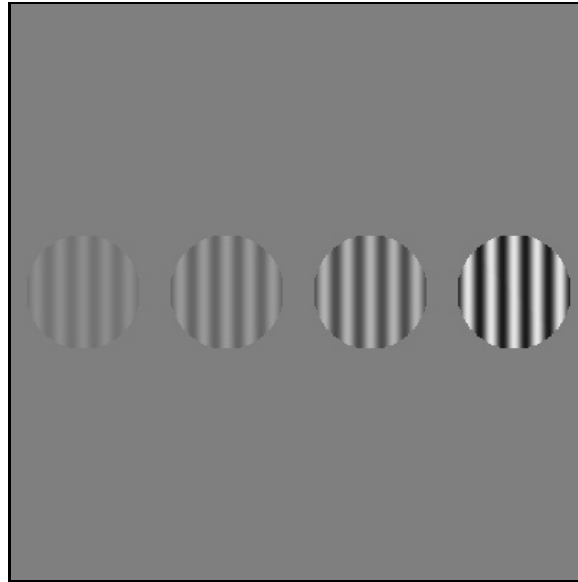
for i_grating in range(4):
    grating.contrast = contrasts[i_grating]
    grating.pos = [grating_hpos[i_grating], 0]

    grating.draw()

win.flip()

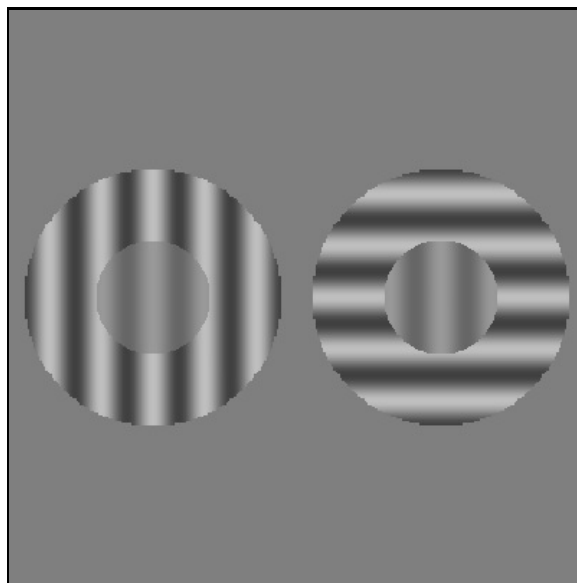
psychopy.event.waitKeys()

win.close()
```



Exercises

1. Another type of mask is referred to as a "raised cosine", termed "raisedCos" in psychopy. What does it look like? When do you think this might be a useful mask to use?
2. Are the masked regions actually "drawn"? In other words, are the blank regions outside a circular mask drawn in grey, or are they somehow transparent? Try drawing two gratings, each with circular masks, such that the square region of one is overlapping the square region of the other. Do you see the effect of a grey square, or are areas outside the mask transparent?
3. Orientation-dependent surround suppression is a perceptual illusion where the contrast of a central grating appears lower when it is surrounded by a grating of the same orientation relative to when it is surrounded by a grating of the orthogonal orientation. Use your knowledge of gratings to recreate the illusion, something like what is shown below:



[Back to top](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. **Drawing—shapes**
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Drawing—shapes

Objectives

- Be able to generate and display stimuli formed from a variety of shapes (lines, rectangles, circles).

Screencast

Using psychopy, we can also easily draw a variety of shapes.

Lines

Perhaps the most simple "shape" stimulus is a line, which we can create in psychopy using `Line`. The two critical arguments to create a line are `start` and `end`, which are each two-item lists that specify the x and y coordinates of the start and end points of the line, respectively.

For example, to draw a line from the bottom left to the top right of our screen, we could do:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

line = psychopy.visual.Line(
    win=win,
    units="pix",
    lineColor=[-1, -1, -1]
)

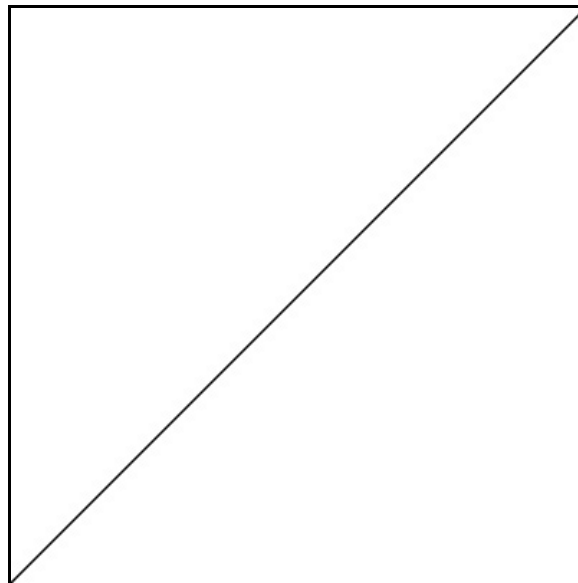
line.start = [-200, -200]
line.end = [+200, +200]

line.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



We can also change the colour of the line using `lineColor` and its width using `lineWidth`:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

line = psychopy.visual.Line(
    win=win,
    units="pix",
    lineColor=[-1, -1, -1]
)

line.start = [-200, -200]
line.end = [+200, +200]
```

```

line.lineColor = [-1, -1, 1]
line.lineWidth = 5

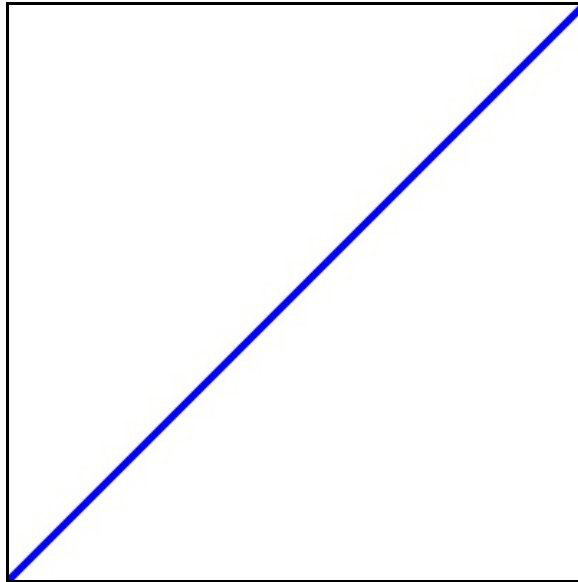
line.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

```



Many visual illusions can be created using simple line arrangements. For example, by drawing a few lines we can create a Ponzo illusion, where the upper horizontal bar looks a bit bigger than the lower horizontal bar:

```

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

line = psychopy.visual.Line(
    win=win,
    units="pix",
    lineColor=[-1, -1, -1]
)

bar_horiz_offset = 20
bar_vert_offset = 80

for bar_offset in [-1, +1]:

    line.start = [-bar_horiz_offset, bar_vert_offset * bar_offset]
    line.end = [+bar_horiz_offset, bar_vert_offset * bar_offset]

    line.draw()

pers_far_horiz_offset = 150
pers_near_horiz_offset = 10
pers_vert_offset = 140

for pers_offset in [-1, +1]:

    line.start = [pers_far_horiz_offset * pers_offset, -pers_vert_offset]
    line.end = [pers_near_horiz_offset * pers_offset, +pers_vert_offset]

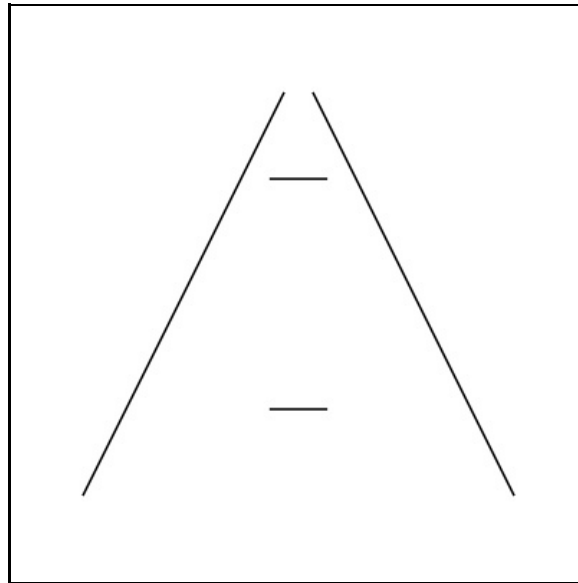
    line.draw()

win.flip()

psychopy.event.waitKeys()

```

```
win.close()
```



Rectangles

We can also create rectangles, using psychopy's `Rect`. The two critical arguments are the `height` and `width` of the rectangle. We can also set a `fillColor`.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

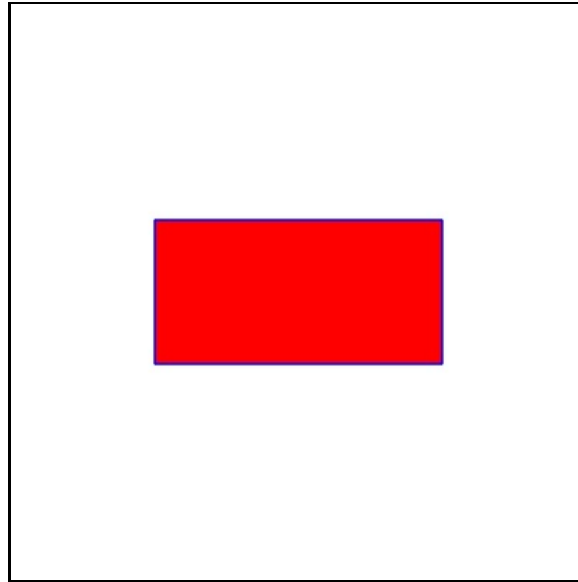
rect = psychopy.visual.Rect(
    win=win,
    units="pix",
    width=200,
    height=100,
    fillColor=[1, -1, -1],
    lineColor=[-1, -1, 1]
)

rect.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



A class of visual stimuli that makes heavy use of rectangles are the so-called *Mondrian* patterns. In such a stimulus, a large number of rectangles of varying widths and heights are drawn at random positions—which ends up producing a pleasant variegated stimulus:

```
import random

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

rect = psychopy.visual.Rect(win=win, units="pix")

n_rect = 500

for i_rect in range(n_rect):

    rect.width = random.uniform(10, 100)
    rect.height = random.uniform(10, 100)

    rect_color = random.uniform(-1, 1)
    rect.fillColor = rect_color
    rect.lineColor = rect_color

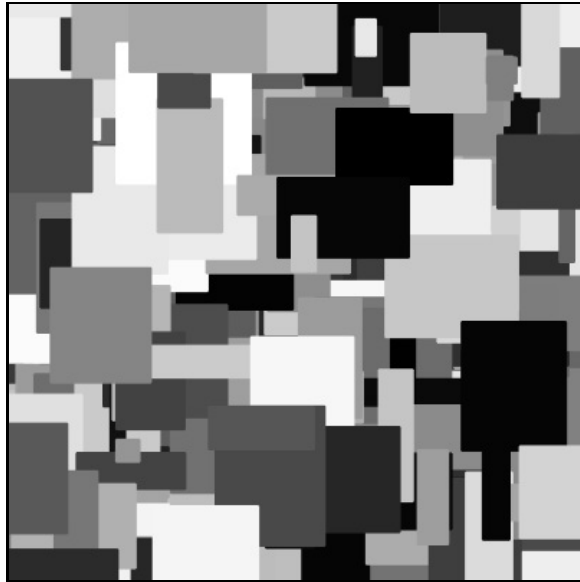
    rect.pos = [
        random.uniform(-200, 200),
        random.uniform(-200, 200)
    ]

    rect.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```

Circles

Circles can also be useful, and are created by psychopy's `Circle`. The critical parameter for a circle is its `radius`.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

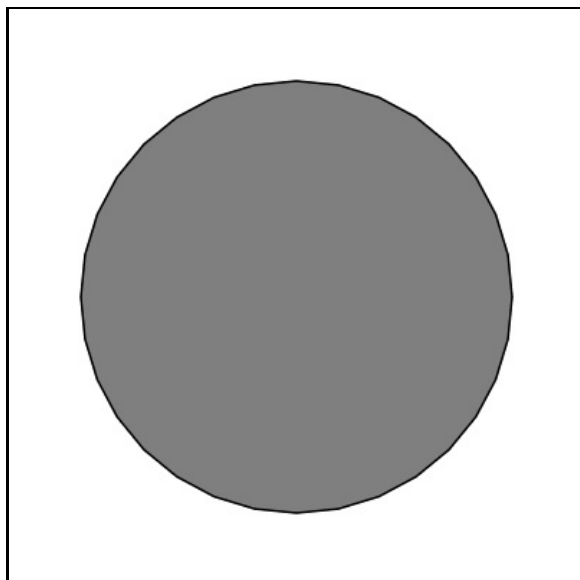
circle = psychopy.visual.Circle(
    win=win,
    units="pix",
    radius=150,
    fillColor=[0, 0, 0],
    lineColor=[-1, -1, -1]
)

circle.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



Notice though that the edges of the circle are not all that smooth. That is because the circle needs to be approximated

with a set of points, and psychopy uses 32 by default. We can increase that a bit to give smoother circles (at the cost of more computation and storage requirements):

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

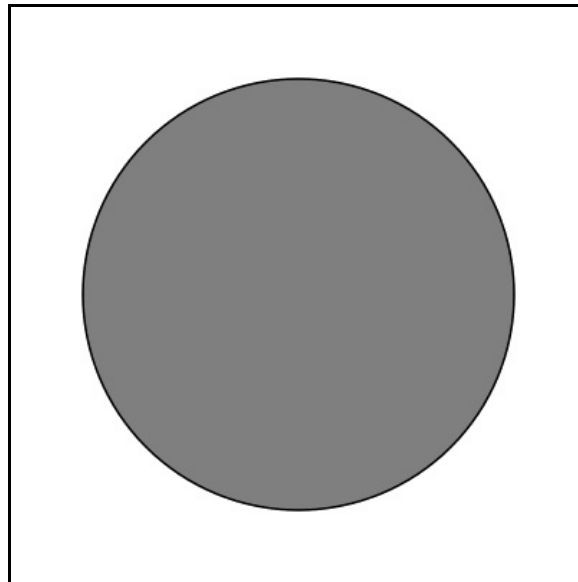
circle = psychopy.visual.Circle(
    win=win,
    units="pix",
    radius=150,
    fillColor=[0, 0, 0],
    lineColor=[-1, -1, -1],
    edges=128
)

circle.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



That looks nicer. Keeping up with the tradition of this lesson, we can create a compelling size illusion (called the Ebbinghaus illusion) using a few circles:

```
import psychopy.visual
import psychopy.event
import psychopy.misc

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

circle = psychopy.visual.Circle(
    win=win,
    units="pix",
    fillColor=[-1] * 3,
    lineColor=[-1] * 3,
    edges=128
)

# 'test' circles
circle.radius = 12
```

```

test_offset = 100

for offset in [-1, +1]:

    circle.pos = [test_offset * offset, 0]

    circle.draw()

# 'surround' circles
surr_thetas = [0, 72, 144, 216, 288]
surr_r = 50

for i_surr in range(len(surr_thetas)):

    [surr_pos_x, surr_pos_y] = psychopy.misc.pol2cart(
        surr_thetas[i_surr],
        surr_r
    )
    surr_pos_x = surr_pos_x + test_offset

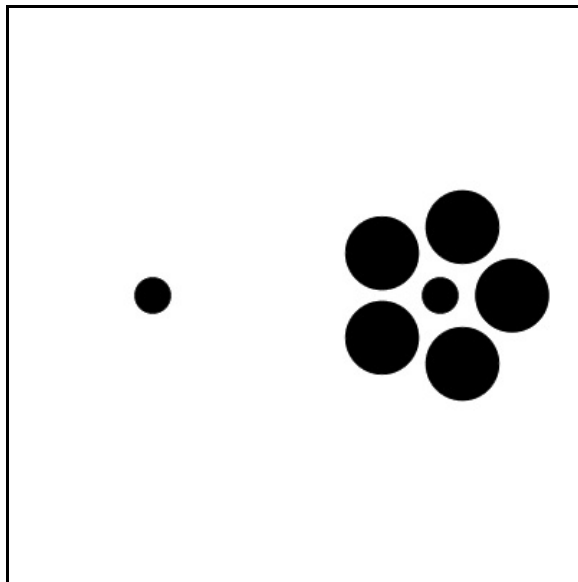
    circle.pos = [surr_pos_x, surr_pos_y]
    circle.radius = 25
    circle.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

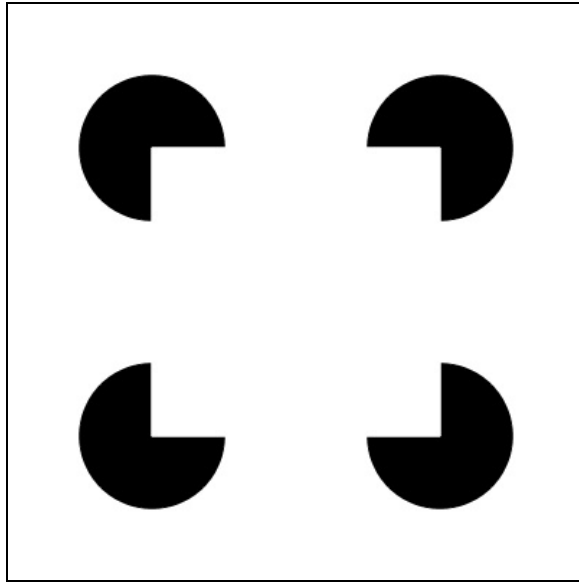
```



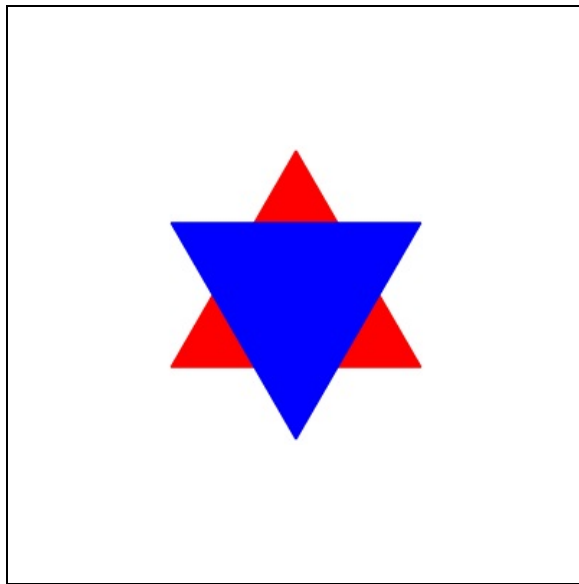
Tip: Note the use of a new function in the above code, `psychopy.misc.pol2cart`. Often when creating visual stimuli it can be more intuitive to work in polar coordinates (polar angle, radius) than cartesian coordinates (x, y). We can use this function to easily convert from polar to cartesian.

Exercises

1. Use your knowledge of shape drawing to create the Kanizsa illusory square stimulus shown below (hint: you don't need to draw any more than five shapes).



2. Investigate `psychopy.visual.Polygon`, and use it to create the stimulus shown below (Hint: do shapes have an `ori` parameter?)



[Back to top](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. **Drawing—images**
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Drawing—images

Objectives

- Be able to display stimuli formed from images stored in files on disk.
- Understand the concept of transparency and its implications for drawing stimuli.
- Know how to create screenshots of the stimulus window.

Screencast

Drawing images from files on disk

While most vision science experiments involve stimuli that are generated rather than those that are loaded from images, there are still situations in which drawing images from files that are stored on disk is required. In psychopy, we can use `ImageStim`.

For example, we might want to draw the [UNSW logo](#), which we have saved to disk as a file called `UNSW.png`. To draw this in Python code, we could do:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

img = psychopy.visual.ImageStim(
    win=win,
    image="UNSW.png",
    units="pix"
)

img.draw()

win.flip()

psychopy.event.waitKeys()
```



We could also display the image in a larger size, although this would entail an inevitable loss in quality. This gets a bit complicated because we might not necessarily know the size of the image—we just know that we want to scale its size. We can use the image's `size` property to sensibly increase the size of the image that we show:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

img = psychopy.visual.ImageStim(
    win=win,
    image="UNSW.png",
    units="pix"
)

size_x = img.size[0]
size_y = img.size[1]

img.size = [size_x * 1.5, size_y * 1.5]
```

```
img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



Transparency and masking

You may notice in the above examples that the "background" in the image we display seems to match the colour of our window. This is because the image file has embedded in it the desired "transparency" of each pixel, which is respected by psychopy. We can see this more clearly if we draw the image after we have drawn a rectangle:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

rect = psychopy.visual.Rect(
    win=win,
    width=200,
    height=100,
    pos=[0, -50],
    fillColor=[1] * 3
)

rect.draw()

img = psychopy.visual.ImageStim(
    win=win,
    image="UNSW.png",
    units="pix"
)

img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



In addition to such masking, we can also set the overall transparency (also known as "opacity") of many stimulus types in psychopy using the `opacity` parameter. This means that successive drawing operations are blended with what has previously been drawn rather than completely overwriting them. For example, we could set the transparency of the image to 50%—notice how it is blended with other elements in the window:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

rect = psychopy.visual.Rect(
    win=win,
    width=200,
    height=100,
    pos=[0, -50],
    fillColor=[1] * 3
)

rect.draw()

img = psychopy.visual.ImageStim(
    win=win,
    image="UNSW.png",
    units="pix"
)

img.opacity = 0.5

img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```




Saving screenshot images

Finally, we will consider how we can take a "screenshot" of what we have drawn using psychopy. This is often a useful procedure, as it allows us to make figures in articles and reports on our research that give a sense of what stimuli participants were viewing.

We can save screenshots easily in psychopy via a two-step process. First, once we have flipped our window into a state that we would like to save, we can use the command `win.getMovieFrame()` to tell psychopy to save the current state of the window. Then, we use the command `win.saveMovieFrames(img_path)` to save this state to a file on disk, where `img_path` is the name of the file that we want to save it to. For example:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

rect = psychopy.visual.Rect(
    win=win,
    width=200,
    height=100,
    pos=[0, -50],
    fillColor=[1] * 3
)

rect.draw()

img = psychopy.visual.ImageStim(
    win=win,
    image="UNSW.png",
    units="pix"
)

img.opacity = 0.5

img.draw()

win.flip()

win.getMovieFrame()
win.saveMovieFrames("unsw_logo_blend_example.png")

psychopy.event.waitKeys()

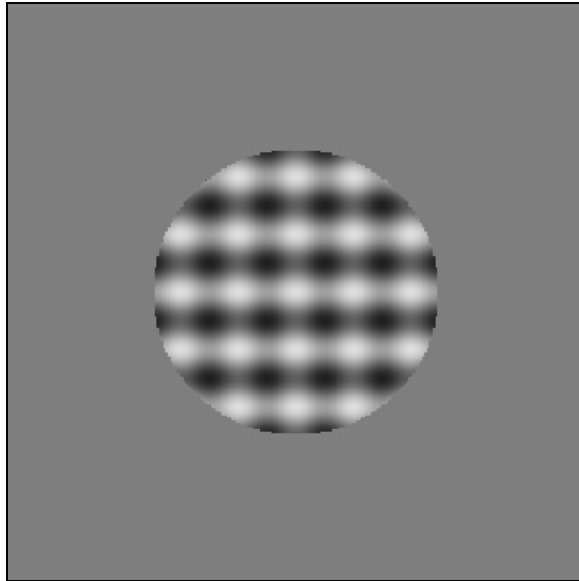
win.close()
```

Exercises

1. Create an image of your own using a drawing program such as Paint or Photoshop. Save it to disk, and then

write code to display it.

2. A "plaid" can be formed from the sum of two sinusoidal gratings of orthogonal orientations (i.e. 90 degrees apart). As we have seen though, if we simply drew one grating after the other the second one would replace the first. Use the `opacity` parameter to approximate plaid formation and produce something like below:



[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. **Drawing—dots**
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Drawing—dots

Objectives

- Be able to create and manipulate stimuli formed from a number of individual elements.

Screencast

Patterns formed from a number of small elements ("dots", typically circles, Gaussians, or squares) are versatile stimuli that are frequently used in vision research. In psychopy, we typically create dot stimuli using `ElementArrayStim`.

The key point about `ElementArrayStim` is that many of its parameters accept a list, rather than only a single value. This

allows us to conveniently create a stimulus from many individual elements. The `xys` is a critical parameter that specifies the position of each element. For example, we can draw a set of randomly-positioned dots by:

```
import random

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

n_dots = 200

dot_xys = []

for dot in range(n_dots):

    dot_x = random.uniform(-200, 200)
    dot_y = random.uniform(-200, 200)

    dot_xys.append([dot_x, dot_y])

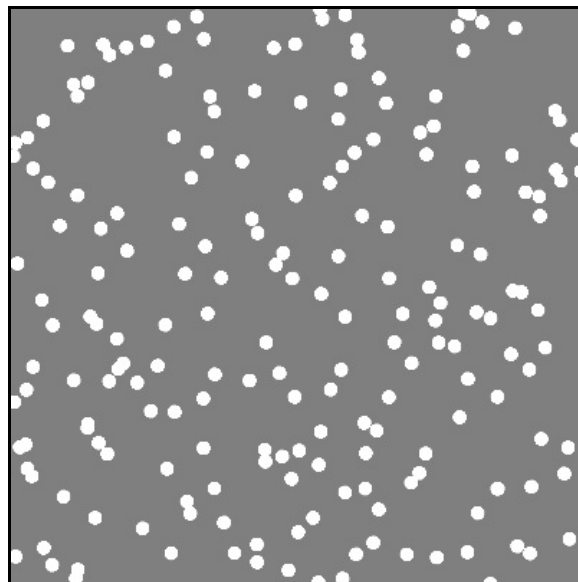
dot_stim = psychopy.visual.ElementArrayStim(
    win=win,
    units="pix",
    nElements=n_dots,
    elementTex=None,
    elementMask="circle",
    xys=dot_xys,
    sizes=10
)

dot_stim.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



There are a few things to note about the code above:

- We specify `xys` as a list of lists; the "outer" list has a number of elements that matches the number of dots. Each element is itself a list, with two elements—the x and y location of that particular dot.
- Note the use of `elementTex` here. By setting it to the special Python value `None`, we are telling psychopy that we want the "texture" of each dot to just be uniform white. Then, by setting `elementMask` to `"circle"`, we are able to define the shape of each individual dot.
- Notice how the size of the dots is specified by the parameter `sizes`, rather than the typical `size`. That means that we could provide a list with a length corresponding to the number of dots, with each element of the list specifying

the size of that particular dot. By just giving a single number here, we are telling psychopy to use this value for all of the dots.

The `ElementArrayStim` doesn't just have to be used for shape-based dots, though. By setting the `elementTex` and `elementMask` parameters, we have great flexibility in what comprises our stimulus. For example, we could use lots of little gratings:

```
import random

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

n_dots = 200

dot_xys = []

for dot in range(n_dots):

    dot_x = random.uniform(-200, 200)
    dot_y = random.uniform(-200, 200)

    dot_xys.append([dot_x, dot_y])

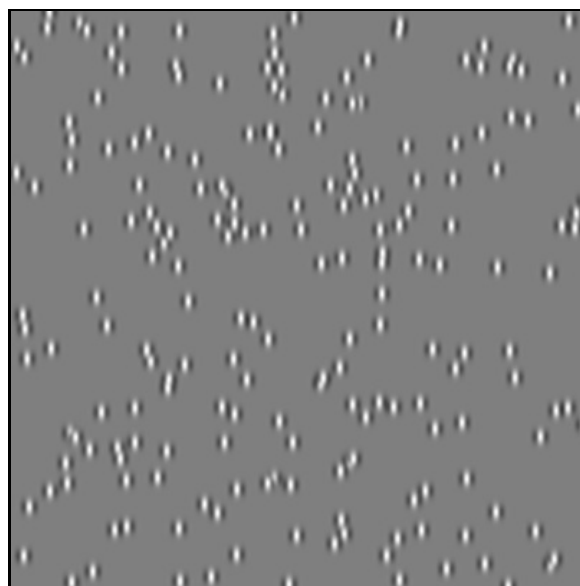
dot_stim = psychopy.visual.ElementArrayStim(
    win=win,
    units="pix",
    nElements=n_dots,
    elementTex="sin",
    elementMask="gauss",
    sfs=5.0 / 2.5,
    xys=dot_xys,
    sizes=20
)

dot_stim.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



Tip: In `ElementArrayStim`, the `sfs` parameter when `units="pix"` is the number of cycles per element, rather than the number of cycles per pixel.

Perhaps with random orientations:

```
import random

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

n_dots = 200

dot_xys = []
dot_oris = []

for dot in range(n_dots):

    dot_x = random.uniform(-200, 200)
    dot_y = random.uniform(-200, 200)

    dot_xys.append([dot_x, dot_y])

    dot_oris.append(random.uniform(0, 180))

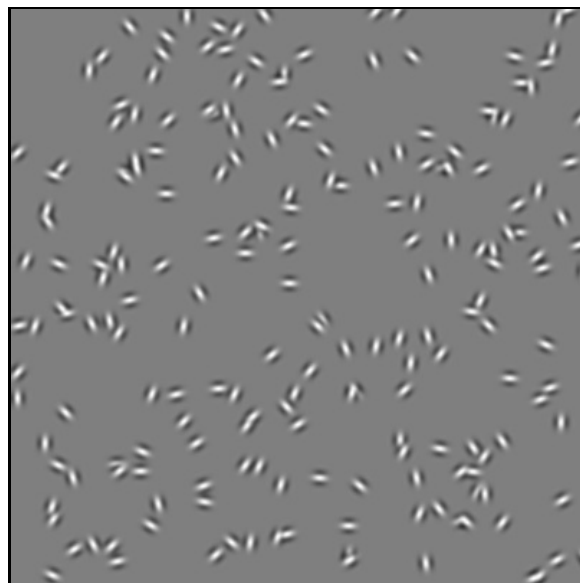
dot_stim = psychopy.visual.ElementArrayStim(
    win=win,
    units="pix",
    nElements=n_dots,
    elementTex="sin",
    elementMask="gauss",
    sfs=5.0 / 2.5,
    xys=dot_xys,
    sizes=20,
    oris=dot_oris
)

dot_stim.draw()

win.flip()

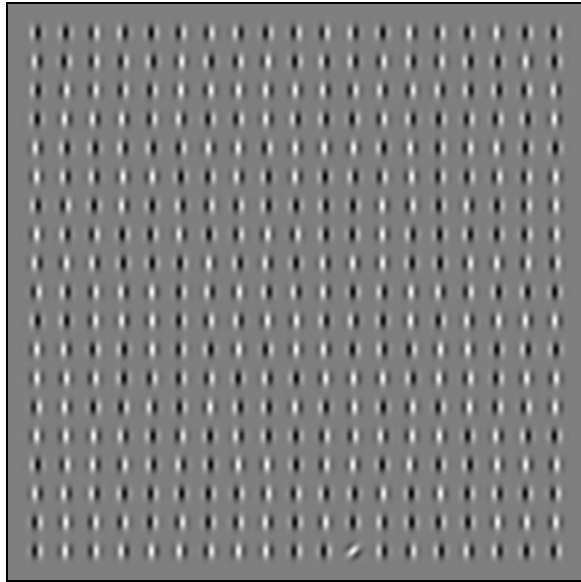
psychopy.event.waitKeys()

win.close()
```



Exercises

1. Create an "odd-one-out" stimulus such as shown below (note that the phases are randomised):



[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. **Temporal dynamics**
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Temporal dynamics

Objectives

- Be able to use clocks to track time during program execution.
- Understand the concept of the monitor refresh rate and its implications for drawing dynamic stimuli.
- Be able to structure code to draw moving stimuli and collect responses.

Screencast

In the previous lessons, we have only considered situations in which the stimuli are static and there is a simple

timecourse in waiting for a response from the observer. However, in vision science experiments we often want to show moving or otherwise dynamic stimuli and with a complex presentation and response schedule. First, we need to investigate how we handle time in Python and psychopy.

Time and clocks

One important way in which we can manipulate the temporal dynamics of our programs is to be able to track time. We can do this in psychopy by creating a `psychopy.core.Clock()`, which has a `getTime()` function that tells us how many seconds have elapsed since the clock was created.

```
import psychopy.core

clock = psychopy.core.Clock()

for iteration in range(2):

    psychopy.core.wait(1.0)

    print clock.getTime()
```

```
1.0524930954
2.0533618927
```

We can use such a clock as a simple way of presenting a stimulus for a desired duration. For example, we could show a grating for 500ms by:

```
import psychopy.visual
import psychopy.event
import psychopy.core

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

clock = psychopy.core.Clock()

text = psychopy.visual.TextStim(win=win)

grating = psychopy.visual.GratingStim(
    win=win,
    size=[200, 200],
    mask="circle",
    units="pix",
    sf=5.0 / 200.0
)

text.text = "Press any key to show the grating"

text.draw()

win.flip()

psychopy.event.waitKeys()

clock.reset()

while clock.getTime() < 0.5:
    grating.draw()
    win.flip()

text.text = "Press any key to finish"

text.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```

The key lines are highlighted in the above code. First, we reset the clock to zero. Then, we use a `while` loop to draw the grating and update the window for as long as the time that has elapsed (which we find out by calling

`clock.getTime()` is less than how long we want to show the grating for (500ms, or 0.5 seconds).

The above method is fine for experiments in which precise timing is not required. Uncertainties in timing that arise using this approach come about, in part, due to the way that the window is updated. Have a think about the `while` loop above—how frequently do you think the grating will be drawn and the window updated? Typically, this is limited by the monitor's "refresh rate", which specifies how often it updates. This is measured in hertz (Hz), and 60Hz is a common rate for LCD monitors.

We can have a look at our refresh rate by asking psychopy to record the intervals between successive `flip` operations. As seen below, on my desktop computer that is used to generate these lessons the refresh interval is about 0.016 seconds, or about 60Hz.

```
import psychopy.visual
```

```
win = psychopy.visual.Window(  
    size=[400, 400],  
    units="pix",  
    fullscr=False  
)
```

```
win.recordFrameIntervals = True
```

```
for frame in range(5):  
    win.flip()
```

```
print win.frameIntervals
```

```
[0.016705989837646484, 0.016644001007080078, 0.016719818115234375,  
0.016662120819091797]
```

If we are confident that the precise refresh rate is known and that it is stable (neither of which can happen without specialised testing), we can instead present our stimuli for a given number of "flips" (known as a given number of "frames"). For example, if we know that our monitor updates at 60 frames per second and we want to draw a grating for 500ms, we can draw 30 frames:

```
import psychopy.visual  
import psychopy.event
```

```
win = psychopy.visual.Window(  
    size=[400, 400],  
    units="pix",  
    fullscr=False  
)
```

```
text = psychopy.visual.TextStim(win=win)
```

```
grating = psychopy.visual.GratingStim(  
    win=win,  
    size=[200, 200],  
    mask="circle",  
    units="pix",  
    sf=5.0 / 200.0  
)
```

```
text.text = "Press any key to show the grating"
```

```
text.draw()
```

```
win.flip()
```

```
psychopy.event.waitKeys()
```

```
for frame in range(30):  
    grating.draw()  
    win.flip()
```

```
text.text = "Press any key to finish"
```

```
text.draw()
```

```
win.flip()
```

```
psychopy.event.waitKeys()
```

```
win.close()
```

Drawing dynamic stimuli

Now that we know about timing and clocks, let's use them to draw a moving stimulus. What we'd like to do is draw a grating that changes phase such that it moves through two complete cycles in one second. We can do that via code such as (note that there are a few new elements to this code—we will go through it below):

```
import numpy as np

import psychopy.visual
import psychopy.event
import psychopy.core

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    size=[200, 200],
    mask="circle",
    units="pix",
    sf=5.0 / 200.0
)

clock = psychopy.core.Clock()

keep_going = True

while keep_going:

    grating.phase = np.mod(clock.getTime() / 0.5, 1)

    grating.draw()

    win.flip()

    keys = psychopy.event.getKeys()

    if len(keys) > 0:
        keep_going = False

win.close()
```

The first new thing is the structure of our loop. We first set a boolean variable `keep_going` to `True`. We will use this to indicate whether the user has pressed a key to indicate that they'd like the program to finish. Because we don't know when this will be, we use a `while` loop to keep repeating until `keep_going` becomes `False`.

The second new thing is the way we are setting the phase of the grating. Recall that the phase is specified by a value that is between 0 and 1. Let's start by thinking about what would happen if we wanted the grating to move through one complete cycle in one second (rather than the two cycles per second we want it to have eventually). We can think of that as wanting the grating's phase to change from 0 to 1 across the course of 1 second of time. Hence, we could simply divide the time, in seconds, by 1 in order to determine the phase. However, remember that the phase needs to be between 0 and 1; as soon as the time is greater than one second, the phase value will be greater than 1. We can fix this by using the `mod` command, which effectively "wraps" the value back around on itself (e.g. $1.2 \bmod 1$ is 0.2, $2.7 \bmod 1$ is 0.7, etc.). The way it does this is by dividing the first number by the second and returning the remainder.

Having established the necessary phase at a given point in time, we use it to update the phase of the grating stimulus. Because this statement is operating within the `while` loop, it is being frequently evaluated and so the grating is frequently changing phase. This change in phase over time gives the visual appearance of a smoothly moving grating.

The final new thing is how we handle the keyboard. In the past, we have used the `psychopy.event.waitKeys()` function, which halts execution until a key is pressed. We can't use that here because we wouldn't be able to update our stimulus if we're waiting for a keypress. Instead, we use the `psychopy.event.getKeys()` function, which probes the state of the keyboard at that precise moment and returns immediately a list of the keys that are being pressed. If a key has been pressed, this list will have a length (`len`) that is greater than zero—in this case, we set our `keep_going` variable to `False` so that we can exit out of the `while` loop.

Controlling the trial schedule

We can use timing and clocks to control the temporal schedule of a given trial in an experiment. For example, we may want to start the trial with 500ms of no stimulus, then show a stimulus for 500ms, then wait for a participant to respond. Furthermore, we wish to require a minimum of 2 seconds between trials. We can do that via:

```
import numpy as np

import psychopy.visual
import psychopy.event
import psychopy.core

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    size=[200, 200],
    mask="circle",
    units="pix",
    sf=5.0 / 200.0
)

clock = psychopy.core.Clock()

n_trials = 2
pre_duration_s = 0.5
stim_duration_s = 0.5
min_iti = 2.0

for trial in range(n_trials):

    clock.reset()

    # wait until the 'pre' time has passed
    while clock.getTime() < pre_duration_s:
        win.flip()

    while clock.getTime() < pre_duration_s + stim_duration_s:
        grating.draw()
        win.flip()

    # clear the window
    win.flip()

    keys = psychopy.event.waitKeys()

    while clock.getTime() < min_iti:
        win.flip()

win.close()
```

Exercises

1. Expand the circular dots demonstration in the [Drawing—dots](#) lesson to make the dots move horizontally at a speed of 0.5 pixels per frame. Hint: you may want to think about using the `mod` function to stop your dots from disappearing!
2. The example in this lesson in "Controlling the trial schedule" implements a single-interval forced-choice design (i.e. one stimulus was shown, and a response collected). Your task is to convert it into a temporal two-alternative forced choice design; each trial has 500ms of blank, then 500ms of a horizontal grating, then 500ms of blank, then 500ms of vertical grating, then a response collection period. The minimum inter-trial-interval is 3 seconds.
3. Rather than presenting the grating stimulus with an abrupt onset and offset, you decide to instead linearly ramp its contrast in and out. Amend the trial stimulus presentation to use frames rather than time and implement the contrast ramping. Hint: you can use `ramp = np.linspace(0, 1, 15).tolist() + np.linspace(1, 0, 15).tolist()` to give you a contrast ramp.

[Back to top](#)

Programming for Psychology in Python

Vision Science

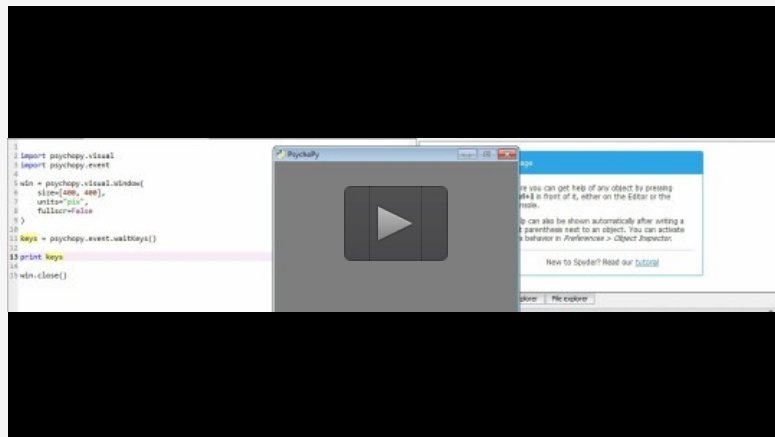
0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. **Collecting responses**
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Collecting responses

Objectives

- Be able to collect responses from the keyboard, knowing which key was pressed and when it was pressed.

Screencast



We have already encountered the way in which we can collect participant responses via our use of `psychopy.event.waitKeys()` and `psychopy.event.getKeys()`. Here, we will take a closer look at these functions and

how we can use them in vision science experiments.

Knowing what keys were pressed

We often want to know *which* key was pressed, not just that any of the keys were pressed. The "key" functions in psychopy return a list of strings, where each item in the list is a string representation of the key that was pressed. For example, if we run the following and press the "d" key, we can see that the item in the list is "d":

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

keys = psychopy.event.waitKeys()

print keys

win.close()
```

```
['d']
```

Tip: Why is `keys` a list, rather than just a string? This is to account for a situation in which multiple keys are pressed simultaneously.

The string that represents the key that was pressed is straightforward for most of the keys. However, we often want to use the "arrow" keys, and it is not immediately obvious what the string representation will be. Let's have a look by running the above again but pressing the "left arrow" key:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

keys = psychopy.event.waitKeys()

print keys

win.close()
```

```
['left']
```

As you can see, the arrow keys correspond to "left", "right", "up", and "down". If you are unsure what the string representation of a particular key is, you can use a method like that shown above to find out.

Restricting the available keys

During an experiment, a participant usually only has a narrow range of keys that are meaningful. To prevent an accidental keypress prematurely ending a call to `waitKeys`, we can provide a `keyList` argument that restricts the keys that it "listens" to. For example, if we know that participants can only respond by pressing the left or right arrow key, it is often sensible to use:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)
```

```
keys = psychopy.event.waitKeys(keyList=["left", "right"])

win.close()
```

This way, the only way that `waitKeys` will return is if the left or right arrow key is pressed—it will keep waiting if another key, such as the spacebar, is pressed.

Determining when a key was pressed

It can also be useful know the time that a keypress was made (such as in studies of reaction time, though care needs to be taken if precise time estimates are required). We can easily obtain this information by providing a `timeStamped` argument with a timing clock:

```
import psychopy.visual
import psychopy.event
import psychopy.core

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

clock = psychopy.core.Clock()

keys = psychopy.event.waitKeys(timeStamped=clock)

print keys

win.close()
```

```
[('return', 0.3359801769256592)]
```

You can see that what is being returned to us now is a list of lists (technically a list of tuples, which are similar to lists), where each item in the list is a pairing of the key that was pressed and the time it was pressed, relative to the clock timing that was provided to the function.

Exercises

1. Investigate the string that is returned when a number is pressed on the keypad versus when the same number is pressed on the horizontal row of keys.
2. Does caps lock have an effect on the string representation of keys?

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. **Providing input**
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Providing input

Objectives

- Be able to construct a simple Graphical User Interface to receive input at the beginning of program execution.

Screencast

Typically when we run an experiment we would like it to be as close to identical for all participants (within a common group, at least) as possible. However, there are often aspects that are specific to a particular time at which you are

executing your program. For example, you might want to provide a participant identifier in order to save the data in a particular location, or you might want to give a repeat number, or perhaps a condition number, etc.

A straightforward way to do this in psychopy is to build a simple GUI (Graphical User Interface) that is displayed when the program starts. The user can then input the required information, which is then available to the rest of the program.

We begin creating such a GUI by using `psychopy.gui.Dlg()`. After creating it, we add various elements to it before using its `show` function. Without any elements yet, it would look something like:

```
import psychopy.gui

gui = psychopy.gui.Dlg()

gui.show()
```

Let's add a field so that the user can provide a subject ID and a run number. I'll enter "p1000" as the subject ID and "1" as the run number.

```
import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Condition Num:")

gui.show()

print gui.data
```

```
[u'p1000', u'1']
```

As you can see, the `gui.data` property holds a list where the items are the GUI elements, in the order they were added.

Tip: Don't worry about the "u" characters in front of the quotation marks in the output. You can just treat them like regular strings.

Also note that information like the condition number is represented as a string, whereas we may want it as an integer. It is straightforward to convert a string to an integer by using the `int` function. For example:

```
import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Condition Num:")

gui.show()

subj_id = gui.data[0]
cond_num = int(gui.data[1])

print subj_id, cond_num
```

```
p1000 1
```

Exercises

1. Use a GUI to get the subject ID and condition number from a user, as above. Then, use this information to construct a filename that you could save the data to, in the format "subj id"_experiment_cond_"cond num".tsv (for example, p1000_exp_1.tsv).

[Back to top](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. **Saving data**
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Saving data

Objectives

- Be able to check for the existence of files on disk.
- Understand how to represent data to facilitate saving to disk.
- Be able to save data to a file on disk.

Screencast

The typical purpose of running an experiment is to generate some data that we can use to address our question of

interest. Hence, it is important that we are able to store the data generated by our experiments to a file on disk.

Creating and checking a file location

The first step in saving data from an experiment is knowing where to save the data to. A common scenario is to save the data from each session (subject and repeat, perhaps condition) to a separate file. This approach maximises the flexibility for subsequent analyses. We may end up with a file name such as `p1000_exp_cond_1_rep_1.tsv`.

A useful step at this point is to check that a file with this name doesn't already exist—it is very frustrating to lose data because it was overwritten! This is also best done right at the start of an experiment, before any data is collected, so that the impact of a wrong filename is minimised. We can check whether the file already exists by using

`os.path.exists`:

```
import os

data_path = "p1000_exp_cond_1_rep_1.tsv"

data_path_exists = os.path.exists(data_path)

print data_path_exists
```

```
True
```

If the file already exists, a safe strategy would be to exit the program at this point and tell the user about the problem. One way to do this is to use the `sys.exit` function:

```
import os
import sys

data_path = "p1000_exp_cond_1_rep_1.tsv"

data_path_exists = os.path.exists(data_path)

# we will pretend that it does exist
data_path_exists = True

if data_path_exists:
    sys.exit("Filename " + data_path + " already exists!")
```

```
Filename p1000_exp_cond_1_rep_1.tsv already exists!
```

Organising data output

While the data you collect in an experiment may come in many forms, it is best to work out a way in which the data can be represented in a "tabular" form; that is, consisting of rows and columns. This is the easiest format with which to store data on disk.

For example, say your experiment consists of 10 trials where each trial shows a grating at a particular orientation and records whether the participant pressed the left or right arrow key in response. This could be organised in the data file as 10 rows and 2 columns; each row represents a trial, and the two columns give the grating orientation and the response on that particular trial. For example:

```
import random
import pprint

data = []

for trial in range(10):

    data.append(
        [
            random.uniform(0, 180),
            random.choice(["left", "right"])
        ]
    )

pprint.pprint(data)
```

```
[[112.68930164014616, 'left'],
 [170.5342789211602, 'right'],
 [136.33854586157733, 'left'],
```

```
[161.78775579304477, 'right'],
[31.593363655057743, 'left'],
[17.65579478494715, 'left'],
[107.17299672417734, 'left'],
[130.67136489516145, 'right'],
[145.5624256321937, 'right'],
[3.352576561670164, 'right']]
```

Tip: Note that we are using a "pretty printer" (`import pprint`) to show the contents of `data`, as it makes the tabular organisation clearer.

However, to make it easier to save and load data to disk, we often convert strings into numbers by coding them. For example, rather than having "left" and "right", we might use the numbers 1 and 2 to refer to the keys that were pressed.

```
import random
import pprint

data = []

for trial in range(10):

    data.append(
        [
            random.uniform(0, 180),
            random.choice(["left", "right"])
        ]
    )

pprint.pprint(data)

coded_data = []

for data_row in data:

    if data_row[1] == "left":
        data_row[1] = 1
    elif data_row[1] == "right":
        data_row[1] = 2

    coded_data.append(data_row)

pprint.pprint(coded_data)
```

```
[[15.18777735976749, 'right'],
[165.43315481513574, 'right'],
[35.28877802769985, 'left'],
[175.48884007552917, 'right'],
[49.05223021460455, 'right'],
[115.75315381947878, 'left'],
[6.089203436153845, 'right'],
[150.0066912999065, 'right'],
[74.96452005619209, 'right'],
[138.63258440058678, 'right']]
[[15.18777735976749, 2],
[165.43315481513574, 2],
[35.28877802769985, 1],
[175.48884007552917, 2],
[49.05223021460455, 2],
[115.75315381947878, 1],
[6.089203436153845, 2],
[150.0066912999065, 2],
[74.96452005619209, 2],
[138.63258440058678, 2]]
```

Saving data

Once we have our data assembled in a suitable format (i.e. list of lists, as above), we can save it to disk using the `np.savetxt` function. The first argument is the filename, and the second is the variable containing the data to be saved. We will also specify an optional argument called `delimiter`, which we will set as `"\t"`. This tells the `savetxt` function that we want columns to be separated by a TAB character. This is a common way of storing files, and is reflected in the data's filename (with "tsv" standing for "tab separated values").

```

import random
import numpy as np
import pprint

data = []

for trial in range(10):

    data.append(
        [
            random.uniform(0, 180),
            random.choice([1, 2])
        ]
    )

pprint.pprint(data)

np.savetxt(
    "p1000_exp_cond_1_run_2.tsv",
    data,
    delimiter="\t"
)

```

```

[[58.79398389770012, 2],
 [81.73650533378212, 2],
 [152.74572432027395, 1],
 [81.27465970923136, 1],
 [148.49067748257954, 1],
 [48.67701808868774, 2],
 [132.0739056466208, 1],
 [127.1721314554986, 1],
 [82.63732851570185, 2],
 [121.54101196986754, 2]]

```

We can then go and load the data using `loadtxt` to verify that we are indeed able to do so:

```

import numpy as np
import pprint

data = np.loadtxt(
    "p1000_exp_cond_1_run_2.tsv",
    delimiter="\t"
)

pprint.pprint(data.tolist())

```

```

[[58.79398389770012, 2.0],
 [81.73650533378212, 2.0],
 [152.74572432027395, 1.0],
 [81.27465970923136, 1.0],
 [148.49067748257954, 1.0],
 [48.67701808868774, 2.0],
 [132.0739056466208, 1.0],
 [127.1721314554986, 1.0],
 [82.63732851570185, 2.0],
 [121.54101196986754, 2.0]]

```

Exercises

1. It is often good practice to include a 'header' in the data file that you save that provides some extra information and context to the data. For example, it might be a good place to explain that 1 means left and 2 means right in the example we have used. Modify the example above to save the data with a header by investigating and using the `header` argument to `np.savetxt`.

[Back to top](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. **Putting it all together—a framework and an example**
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Putting it all together—a framework and an example

Objectives

- Be able to use a framework to guide experimental design and implementation.
- Be able to follow a worked example of a complete experiment implemented in Python.

Across the series of previous lessons, we have covered a lot of the mechanics that go into creating a vision science experiment using Python and psychopy. Here, we are going to investigate how we can put all these components together in order to actually implement an experiment.

You now have a set of skills that mean there are very few technical impediments to running the experiment that you want to run. However, a very important thing to keep in mind is that the experience of an experiment is almost completely separate from its implementation. Hence, we need to consider precisely what it is that we want to happen during an experiment.

We are going to investigate a framework that you can use to think about the key components of your experiment. Once these components are in place, there is a very natural translation of their requirements into Python code. Note that these assume that you have already decided on the experiment's question and hypothesis, its independent variable(s), its dependent variable(s), and its design.

We are going to examine the framework in the context of an example experiment. In this experiment, we are interested to understand whether the speed at which an ambiguous figure rotates affects how often its perceptual interpretation switches. We will be using a structure-from-motion stimulus, in which changes in local dot speed give the impression of a rotating cylinder. The independent variable will be the rotational speed of the cylinder (either 0.2 or 0.1 revolutions per second) and the dependent variable will be the frequency at which the percept of the cylinder switches.

We will consider each of the components of the experiment creation framework in turn, building up the code for the example experiment as we go.

1. What information is needed to begin an experiment session?

Here, we need to think about what information will be unique to a particular session. This will almost always include some participant identifier (such as "subj_1"), usually also in combination with a detail specific to that participant (such as the "run"/repeat number or an indicator of the experimental condition).

For the example experiment, we need the participant identifier, the condition number (1 or 2), and the repeat number:

```
import os
import sys

import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Cond. num.:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
cond_num = gui.data[1]
rep_num = gui.data[2]

data_path = subj_id + "_cond_" + cond_num + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")
```

The above should all be familiar from the [Providing input](#) and [Saving data](#) lessons.

2. What data will be recorded from the experiment session?

Here, we think about the nature of the data that the experiment will produce. At this stage, we are thinking about the data at a "high level" rather than specific details. For example, we may want to simply record behavioural responses, or we may want to also track eye movements, or we may want to record EEG signals.

For the example experiment, our requirements are simple—we just need to collect behavioural responses. We prepare to record behavioural data by creating an empty list, which we will expand with data as the experiment progresses.

```
import os
import sys

import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Cond. num.:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
cond_num = gui.data[1]
rep_num = gui.data[2]

data_path = subj_id + "_cond_" + cond_num + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

responses = []
```

3. What sort of stimuli are required?

Here, we need to think about the types of stimuli that will be used in the experiment. We need to do so in a great amount of detail, taking into account aspects like size, position, form, dynamics, etc.

For the example experiment, the primary stimulus will be the structure-from-motion cylinder. We want it to be 200 pixels in height and width, have a dot size of 5 pixels, be made from 1000 dots, with each dot having a Gaussian profile. We want it to rotate around the vertical axis, with the vertical location randomised and the phase of the horizontal motion randomised.

```

import os
import sys

import numpy as np

import psychopy.visual
import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Cond. num.:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
cond_num = gui.data[1]
rep_num = gui.data[2]

data_path = subj_id + "_cond_" + cond_num + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

responses = []

sfm_size_pix = 200
sfm_dot_size_pix = 5
sfm_n_dots = 1000
sfm_dot_shape = "gauss"

if cond_num == "1":
    sfm_speed_rev_per_s = 0.2
elif cond_num == "2":
    sfm_speed_rev_per_s = 0.1
else:
    sys.exit("Unknown condition number")

sfm_y_pos = np.random.uniform(-sfm_size_pix / 2, +sfm_size_pix / 2, sfm_n_dots)

sfm_x_phase = np.random.uniform(0, 2 * np.pi, sfm_n_dots)

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

sfm_stim = psychopy.visual.ElementArrayStim(
    win=win,
    units="pix",
    nElements=sfm_n_dots,
    elementTex=None,
    elementMask=sfm_dot_shape,
    sizes=sfm_dot_size_pix
)

win.close()

```

The above should be mostly familiar from the [Drawing to a window](#) and [Drawing—dots](#) lessons. The specifics on how a structure-from-motion cylinder is created will, however, be new. Here, we first set the vertical position of each dot in the cylinder (`sfm_y_pos`) to a random value. We also set the horizontal position of each dot in the cylinder (`sfm_x_phase`) to a random value, but indirectly. Rather than giving each dot a random position in pixels, we give it a random phase between 0 and 2 pi. This is because we will use a cosine function to transform the phase value into a position value during presentation.

4. What trial sequence will be used?

A typical session will be broken down into a series of trials, where each trial typically probes a particular level of the independent variable(s). Here, we need to decide how many trials will be conducted for each condition and the order in which they are presented.

For the example experiment, a given session (a participant for a particular condition at a particular repeat) only consists of a single trial—so there is no need to do anything in the code.

5. What instructions need to be provided?

It is important to think about what information a participant would need in order to be able to complete the assigned task. Such "instructions" may take the form of one or more sets of on-screen text, or they may require extensive training, practice, and feedback.

For the example experiment, we assume that participants are reasonably familiar with the stimulus and just need to be told how to respond. We will do this via on-screen text, which participants can view for as long as they like before pressing any key to commence the experiment:

```
import os
import sys

import numpy as np

import psychopy.visual
import psychopy.event
import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Cond. num.:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
cond_num = gui.data[1]
rep_num = gui.data[2]

data_path = subj_id + "_cond_" + cond_num + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

responses = []

sfm_size_pix = 200
sfm_dot_size_pix = 5
sfm_n_dots = 1000
sfm_dot_shape = "gauss"

if cond_num == "1":
    sfm_speed_rev_per_s = 0.2
elif cond_num == "2":
    sfm_speed_rev_per_s = 0.1
else:
    sys.exit("Unknown condition number")

sfm_y_pos = np.random.uniform(-sfm_size_pix / 2, +sfm_size_pix / 2, sfm_n_dots)

sfm_x_phase = np.random.uniform(0, 2 * np.pi, sfm_n_dots)

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

sfm_stim = psychopy.visual.ElementArrayStim(
    win=win,
    units="pix",
    nElements=sfm_n_dots,
    elementTex=None,
    elementMask=sfm_dot_shape,
    sizes=sfm_dot_size_pix
)

instructions = psychopy.visual.TextStim(
    win=win,
    wrapwidth=350,
)

instructions.text = """
Press the left arrow key when you see the front surface rotating to the left.\n
Press the right arrow key when you see the front surface rotating to the right.\n
"""
```

```

\n
Press any key to begin.
"""

instructions.draw()
win.flip()

psychopy.event.waitKeys()

win.close()

```

Note the slightly new spin on defining a string in the above (see [Data—strings and booleans](#) in Programming fundamentals for a reminder about strings). Rather than using the typical single quotes ("), we have used three quotes """. This is the same idea, but using three quotes allows the string to be defined across multiple lines. You also may not have yet encountered the special "\n" sequence—it indicates a new line.

Other aspects of the above should be familiar from the [Getting started](#), [Drawing to a window](#), and [Providing input](#) lessons. Something you may not have come across is the `wrapWidth` argument to `TextStim`—this asks psychopy to allow the text that is displayed to be a maximum of 350 pixels wide before moving to the next line.

6. What happens on a given trial?

This is a particularly key component of the experiment, as it defines what the participant actually sees and does. Here, we define precisely what happens on a given trial. This sequence often follows a stereotypical format, such as a 1AFC (single stimulus presentation, then response), spatial 2AFC (simultaneous presentation of two stimuli, then response), and temporal 2AFC (presentation of two stimuli one after another, then response).

For the example experiment, the trial format is straightforward. The stimulus is shown for the trial duration (set to 120 seconds), during which time the participant responds whenever they perceive a change in the subjective appearance of the stimulus.

```

import os
import sys

import numpy as np

import psychopy.visual
import psychopy.event
import psychopy.gui
import psychopy.core

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Cond. num.:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
cond_num = gui.data[1]
rep_num = gui.data[2]

data_path = subj_id + "_cond_" + cond_num + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

responses = []

sfm_size_pix = 200
sfm_dot_size_pix = 5
sfm_n_dots = 1000
sfm_dot_shape = "gauss"

if cond_num == "1":
    sfm_speed_rev_per_s = 0.2
elif cond_num == "2":
    sfm_speed_rev_per_s = 0.1
else:
    sys.exit("Unknown condition number")

sfm_y_pos = np.random.uniform(-sfm_size_pix / 2, +sfm_size_pix / 2, sfm_n_dots)

```

```

sfm_x_phase = np.random.uniform(0, 2 * np.pi, sfm_n_dots)

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

sfm_stim = psychopy.visual.ElementArrayStim(
    win=win,
    units="pix",
    nElements=sfm_n_dots,
    elementTex=None,
    elementMask=sfm_dot_shape,
    sizes=sfm_dot_size_pix
)

instructions = psychopy.visual.TextStim(
    win=win,
    wrapwidth=350,
)

instructions.text = """
Press the left arrow key when you see the front surface rotating to the left.\n
Press the right arrow key when you see the front surface rotating to the right.\n
\n
Press any key to begin.
"""

instructions.draw()
win.flip()

psychopy.event.waitKeys()

duration_s = 120.0

clock = psychopy.core.Clock()

while clock.getTime() < duration_s:

    phase_offset = (clock.getTime() * sfm_speed_rev_per_s) * (2 * np.pi)

    sfm_xys = []

    for i_dot in range(sfm_n_dots):

        dot_x_pos = np.cos(sfm_x_phase[i_dot] + phase_offset) * sfm_size_pix / 2.0
        sfm_xys.append([dot_x_pos, sfm_y_pos[i_dot]])

    sfm_stim.xys = sfm_xys

    sfm_stim.draw()

    win.flip()

win.close()

```

Much of the above will be familiar from the [Temporal dynamics](#) lesson. However, the way in which we implement the structure-from-motion will be new.

No need to worry too much about the specifics, but if you're interested—recall that we have specified the horizontal position of each dot by a phase value between 0 and 2 pi. We first need to work out how much this phase value should be changed, given how much time has elapsed since the start of the experiment. A good way to think about it is if we wanted the dots to go through one complete revolution per second—that is, their phase should have changed by 2 pi units in one second of time. If we instead want it to go through two revolutions per second, the phase will have changed 2 pi units in half a second. Hence, to work out how many phase units we should have advanced at the current time we multiply the time by the desired number of revolutions per second and then by 2 pi.

Tip: Note that the "phase" units here are different to our previous experience with phase in relation to `GratingStim`. In that case, phase was bounded between 0 and 1. Here, the `cos` function represents phase as radians, between 0 and 2 pi. We will quickly exceed 2 pi in the code above, and we could consider using a `mod` function to appropriately wrap the phase to be within 0 and 2 pi. However, since that is handled naturally by the `cos` function, we don't worry about it.

Having calculated how much the phase of the dots should be advanced given the time that has elapsed, we then add it to each dot's initial random phase value and use the `cos` function. The result is a position between -1 and +1, which we then convert into pixels by multiplying by half of the stimulus size.

Tip: In the above code, and in the code in these lessons in general, we have emphasised clarity and readability over performance. However, if we were actually conducting this experiment we would need to either confirm that the computations were sufficiently fast that we were not "dropping frames" (not able to keep up with the refresh rate of the screen) or we would need to use (slightly) more sophisticated techniques to improve performance.

7. What data needs to be recorded for each trial?

It is important to think about what data about each trial needs to be saved. The guiding strategy is that you want to be able to have enough information to re-create the experiment afterwards. While things like participant responses are obvious information that needs to be saved, we also need to be particularly aware of any random components to the experiment. For example, if the order of conditions is randomised across the experiment, this trial sequence will be lost unless the condition information for each trial is saved. As a general rule, it is much better to save everything that you think might be useful later on than it is to not save something.

For the example experiment, we want to save the time (measured from the start of the experiment) at which the participant pressed a key, and which key was pressed. We will code the left arrow key as the number 1 and the right arrow key as the number 2.

```
import os
import sys

import numpy as np

import psychopy.visual
import psychopy.event
import psychopy.gui
import psychopy.core

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Cond. num.:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
cond_num = gui.data[1]
rep_num = gui.data[2]

data_path = subj_id + "_cond_" + cond_num + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

responses = []

sfm_size_pix = 200
sfm_dot_size_pix = 5
sfm_n_dots = 1000
sfm_dot_shape = "gauss"

if cond_num == "1":
    sfm_speed_rev_per_s = 0.2
elif cond_num == "2":
    sfm_speed_rev_per_s = 0.1
else:
    sys.exit("Unknown condition number")

sfm_y_pos = np.random.uniform(-sfm_size_pix / 2, +sfm_size_pix / 2, sfm_n_dots)

sfm_x_phase = np.random.uniform(0, 2 * np.pi, sfm_n_dots)

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
```

```

)

sfm_stim = psychopy.visual.ElementArrayStim(
    win=win,
    units="pix",
    nElements=sfm_n_dots,
    elementTex=None,
    elementMask=sfm_dot_shape,
    sizes=sfm_dot_size_pix
)

instructions = psychopy.visual.TextStim(
    win=win,
    wrapWidth=350,
)

instructions.text = """
Press the left arrow key when you see the front surface rotating to the left.\n
Press the right arrow key when you see the front surface rotating to the right.\n
\n
Press any key to begin.
"""

instructions.draw()
win.flip()

psychopy.event.waitKeys()

duration_s = 120.0

clock = psychopy.core.Clock()

while clock.getTime() < duration_s:

    phase_offset = (clock.getTime() * sfm_speed_rev_per_s) * (2 * np.pi)

    sfm_xys = []

    for i_dot in range(sfm_n_dots):

        dot_x_pos = np.cos(sfm_x_phase[i_dot] + phase_offset) * sfm_size_pix / 2.0
        sfm_xys.append([dot_x_pos, sfm_y_pos[i_dot]])

    sfm_stim.xys = sfm_xys

    sfm_stim.draw()

    win.flip()

    keys = psychopy.event.getKeys(
        keyList=["left", "right"],
        timeStamped=clock
    )

    for key in keys:

        if key[0] == "left":
            key_num = 1
        else:
            key_num = 2

        responses.append([key_num, key[1]])

win.close()

```

This should all be familiar from the [Collecting responses](#) lesson.

8. How should the data be saved for further analysis?

The data that is saved from a given invocation of the experimental code is highly unlikely to be the final format in which it is used. Typically, the data would need to be aggregated across repeats, summarised for a given participant, compared across participants, etc.—it is useful to think about what the most useful data format is to facilitate such future analyses. For example, if you know that your analysis will be performed in SPSS, you might want to think about directly writing an SPSS file.

The most flexible strategy, and the one we adopt here, is to use a simple text-based file format. This uses a tabular representation of the data to save it to a file that can then be imported into a variety of other programs, such as Excel

and SPSS.

For the example experiment, we want to save a text file where each row corresponds to a button press by the participant, and the two columns indicate the key that was pressed and the time that it was pressed.

```
import os
import sys

import numpy as np

import psychopy.visual
import psychopy.event
import psychopy.gui
import psychopy.core

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Cond. num.:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
cond_num = gui.data[1]
rep_num = gui.data[2]

data_path = subj_id + "_cond_" + cond_num + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

responses = []

sfm_size_pix = 200
sfm_dot_size_pix = 5
sfm_n_dots = 1000
sfm_dot_shape = "gauss"

if cond_num == "1":
    sfm_speed_rev_per_s = 0.2
elif cond_num == "2":
    sfm_speed_rev_per_s = 0.1
else:
    sys.exit("Unknown condition number")

sfm_y_pos = np.random.uniform(-sfm_size_pix / 2, +sfm_size_pix / 2, sfm_n_dots)

sfm_x_phase = np.random.uniform(0, 2 * np.pi, sfm_n_dots)

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

sfm_stim = psychopy.visual.ElementArrayStim(
    win=win,
    units="pix",
    nElements=sfm_n_dots,
    elementTex=None,
    elementMask=sfm_dot_shape,
    sizes=sfm_dot_size_pix
)

instructions = psychopy.visual.TextStim(
    win=win,
    wrapWidth=350,
)

instructions.text = """
Press the left arrow key when you see the front surface rotating to the left.\n
Press the right arrow key when you see the front surface rotating to the right.\n
\n
Press any key to begin.
"""
```

```

instructions.draw()
win.flip()

psychoPy.event.waitKeys()

duration_s = 120.0

clock = psychoPy.core.Clock()

while clock.getTime() < duration_s:

    phase_offset = (clock.getTime() * sfm_speed_rev_per_s) * (2 * np.pi)

    sfm_xys = []

    for i_dot in range(sfm_n_dots):

        dot_x_pos = np.cos(sfm_x_phase[i_dot] + phase_offset) * sfm_size_pix / 2.0
        sfm_xys.append([dot_x_pos, sfm_y_pos[i_dot]])

    sfm_stim.xys = sfm_xys

    sfm_stim.draw()

    win.flip()

    keys = psychoPy.event.getKeys(
        keyList=["left", "right"],
        timeStamped=clock
    )

    for key in keys:

        if key[0] == "left":
            key_num = 1
        else:
            key_num = 2

        responses.append([key_num, key[1]])

np.savetxt(data_path, responses, delimiter="\t")

win.close()

```

This should all be familiar from the [Saving data](#) lesson.

Summary

We now have a fully-functioning vision science experiment. We reached this point by first emphasising the distinction between the experimental design/requirements and how the experiment is implemented in Python code. We investigated a framework in which a series of questions about the experiment are considered, and how we can use the responses to guide our process of implementing the experiment.

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. **Putting it all together—another example**
 13. [Exercise solutions](#)
 14. [Extra: Spatial frequency filtering](#)
-

Putting it all together—another example

Objectives

- Be able to follow and understand another example of experiment creation and implementation in Python.

In this lesson, we are going to work through the process of implementing another experiment in Python and psychopy.

Our experiment involves a particular visual illusion in which a completely static stimulus appears to be moving, particularly around the time of an eye movement. Perhaps it was motivated after seeing the cover of the album "[Merriweather Post Pavilion](#)" by [Animal Collective](#), as shown below.



We are interested in knowing what factors in the stimulus lead us to perceive the motion. We have an inkling that the light and dark regions at the edge of each oval are critical to the illusion. Hence, we decide to run an experiment where we manipulate the difference between the intensity of the light and dark regions as our independent variable. As our dependent variable, we will have participants rate the subjective experience of motion on a scale from 1 to 5.

As before, we will work through our experiment framework and build up the code as we go through.

1. What information is needed to begin an experiment session?

We will require a participant identifier and a repeat number. As all the experimental conditions will be experienced during each repeat, there is no need to specify anything to do with conditions.

```
import os
import sys

import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
rep_num = gui.data[1]

data_path = subj_id + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")
```

2. What data will be recorded from the experiment session?

We will be recording behavioural data, consisting of the information about each trial.

```
import os
import sys

import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
rep_num = gui.data[1]

data_path = subj_id + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

exp_data = []
```

3. What sort of stimuli are required?

Now we reach the tricky part—how can we create a stimulus containing the illusion? First, we need to closely examine our demonstration stimulus and determine its key components. On visual inspection of the image above, we can see that it is formed from a grid of green oval shapes. Moving across the columns from left to right, we can see that the orientation of each successive oval is shifted anti-clockwise from the previous oval. Moving up the rows from the bottom, we can see that the orientation of each successive oval is shifted clockwise from the previous oval. Looking at each individual oval, we can also see that one side has a white edge and the other side has a black edge. Overall, the ovals are presented on a dark lightly-textured background of blues and purples.

Now that we have a qualitative understanding the key components of the stimulus, let's make them quantitative so we can specify them in our code. First, let's set the overall stimulus size at 400 pixels. This is probably too small if we were running the experiment "for real", but it will do for our purposes. Next, let's set the background colour to a uniform darkish blue; say `[-1, -1, -0.25]`. Now let's think about each oval; we will give it a lightish green colour, say `[-1,`

0.25, -1], and a size of 10 pixels horizontally and 18 pixels vertically. We will give the line around the outside a width of 3 pixels. Ultimately, we will want to manipulate the intensity of the light and dark edges, but for now let's set them to +1 and -1. Now for the grid, let's have 40 ovals both horizontally and vertically, spaced evenly between 20 and 380 pixels in the window. Finally, let's make the change in orientation of successive ovals 30 degrees anti-clockwise across columns and 30 degrees clockwise across ascending rows.

We will approach this by creating a different set of code to our main experiment that we can use to get a working stimulus without worrying about the other components of the experiment. First, let's specify in code the above stimulus details and prepare a window:

```
import psychopy.visual

bg_colour = [-1, -1, -0.25]

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30
oval_edge_contrast = 1.0
oval_line_width = 3.0

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

win.close()
```

Now let's think about the ovals. The basic shape is straightforward; we can simply use a `Circle` with an unequal aspect ratio:

```
import psychopy.visual

bg_colour = [-1, -1, -0.25]

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30
oval_edge_contrast = 1.0
oval_line_width = 3.0

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

n_edges = 128

oval = psychopy.visual.Circle(
    win=win,
    radius=oval_radius_pix,
    units="pix",
    edges=n_edges,
    fillColor=oval_fill_colour,
)

win.close()
```

The tricky part is getting the outside lines to have different intensities. Here, our strategy is going to be to create two new shapes for each oval; one corresponding to the light edge and one corresponding to the dark edge. To do this, we will use a `ShapeStim`, which requires the coordinates of each vertex to be specified. We could work out the positioning of vertices along the required arc, but it is easier just to extract half the vertices from our oval stimulus, which has a property `verticesPix`. To stop `ShapeStim` producing an enclosed shape, we set the `closeShape` parameter to `False`.

```
import psychopy.visual

bg_colour = [-1, -1, -0.25]
```

```

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30
oval_edge_contrast = 1.0
oval_line_width = 3.0

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

n_edges = 128

oval = psychopy.visual.Circle(
    win=win,
    radius=oval_radius_pix,
    units="pix",
    edges=n_edges,
    fillColor=oval_fill_colour,
)

vertices = oval.verticesPix.tolist()

right_vertices = vertices[:,(n_edges / 2 + 1)]

left_vertices = []

for vertex in right_vertices:
    left_vertices.append([-vertex[0], vertex[1]])

left_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=left_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[-oval_edge_contrast] * 3
)

right_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=right_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[+oval_edge_contrast] * 3
)

win.close()

```

OK, so that will give us one oval with light and dark edges. Now, we want to draw a grid of such ovals, with the appropriate offsets in orientation across rows and columns. To do so, we will loop across columns and then across rows—you can visualise this as starting off in the bottom left hand corner, then moving one spot to the right, then continuing until reaching the end of the row at which point you move to the second row from the bottom and repeat the process.

First, we set the starting orientation of the items in a given row. We do this by multiplying the row index by how much we want the orientation to change across rows (30 degrees, in our example). Then, for each column we add the column offset (-30 degrees, in our example). We then update the position and orientation and draw the stimulus. We can then have a look at it!

```

import psychopy.visual
import psychopy.event

bg_colour = [-1, -1, -0.25]

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30

```

```

oval_edge_contrast = 1.0
oval_line_width = 3.0

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

n_edges = 128

oval = psychopy.visual.Circle(
    win=win,
    radius=oval_radius_pix,
    units="pix",
    edges=n_edges,
    fillColor=oval_fill_colour,
)

vertices = oval.verticesPix.tolist()

right_vertices = vertices[:((n_edges / 2 + 1))]

left_vertices = []

for vertex in right_vertices:
    left_vertices.append([-vertex[0], vertex[1]])

left_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=left_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[-oval_edge_contrast] * 3
)

right_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=right_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[+oval_edge_contrast] * 3
)

offsets = range(-180, 181, n_ovals_per_dim)

row_count = 0

for y_offset in offsets:

    ori = row_count * oval_row_ori_change

    for x_offset in offsets:

        ori = ori + oval_col_ori_change

        for stim in [oval, left_side, right_side]:
            stim.pos = [x_offset, y_offset]
            stim.ori = ori
            stim.draw()

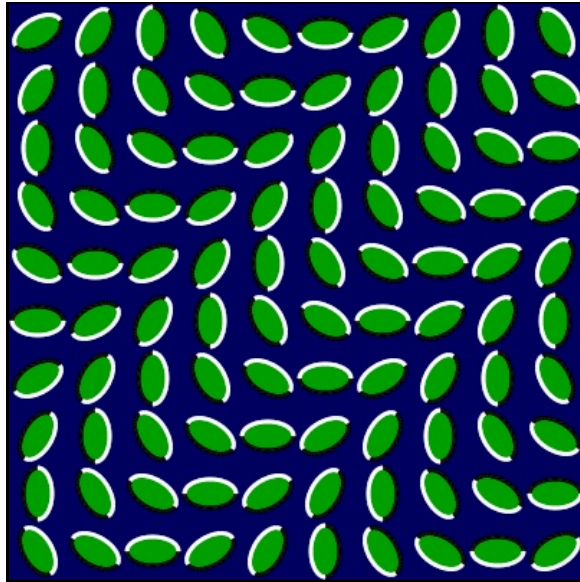
        row_count = row_count + 1

win.flip()

psychopy.event.waitKeys()

win.close()

```



Not quite the same, but close enough. Now, let's fold the stimulus creation into our experiment code, leaving the drawing until later.

```
import os
import sys

import psychopy.visual
import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
rep_num = gui.data[1]

data_path = subj_id + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

exp_data = []

bg_colour = [-1, -1, -0.25]

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30
oval_edge_contrast = 1.0
oval_line_width = 3.0
oval_n_edges = 128

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

oval = psychopy.visual.Circle(
    win=win,
    radius=oval_radius_pix,
    units="pix",
    edges=oval_n_edges,
    fillColor=oval_fill_colour,
)

vertices = oval.verticesPix.tolist()

right_vertices = vertices[:,(oval_n_edges / 2 + 1)]
```

```

left_vertices = []

for vertex in right_vertices:
    left_vertices.append([-vertex[0], vertex[1]])

left_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=left_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[-oval_edge_contrast] * 3
)

right_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=right_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[+oval_edge_contrast] * 3
)

offsets = range(-180, 181, n_ovals_per_dim)

win.close()

```

4. What trial sequence will be used?

Our independent variable in this experiment is the difference of the light and dark edges from mid-grey. We will probe five levels, equally spaced between 0 and 1, with each having 10 trials per repeat. We want the order of trials to be randomised across the course of the repeat. We can achieve the above in code via:

```

import os
import sys
import random

import psychopy.visual
import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
rep_num = gui.data[1]

data_path = subj_id + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

exp_data = []

bg_colour = [-1, -1, -0.25]

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30
oval_edge_contrast = 1.0
oval_line_width = 3.0
oval_n_edges = 128

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

oval = psychopy.visual.Circle(

```

```

    win=win,
    radius=oval_radius_pix,
    units="pix",
    edges=oval_n_edges,
    fillColor=oval_fill_colour,
)

vertices = oval.verticesPix.tolist()

right_vertices = vertices[::(oval_n_edges / 2 + 1)]

left_vertices = []

for vertex in right_vertices:
    left_vertices.append([-vertex[0], vertex[1]])

left_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=left_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[-oval_edge_contrast] * 3
)

right_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=right_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[+oval_edge_contrast] * 3
)

offsets = range(-180, 181, n_ovals_per_dim)

oval_line_contrasts = [0, 0.25, 0.5, 0.75, 1.0]
n_oval_line_contrasts = len(oval_line_contrasts)

n_trials_per_line_contrast = 10

trial_oval_line_contrasts = oval_line_contrasts * n_trials_per_line_contrast
random.shuffle(trial_oval_line_contrasts)

win.close()

```

In the above, we first created a list of the line intensity differences (which we call the "line contrast") for all 50 trials (5 conditions times 10 trials per condition), in non-random order. We then used `random.shuffle` to randomise this list—if we now use the items in turn, the order of conditions will be randomised.

5. What instructions need to be provided?

Here, we will simply instruct the participant of what keys to push and how they relate to their perception. Were we creating a real experiment, we would likely want to create some more involved instructions so that we gain better control of the criteria that participants use to make their ratings.

```

import os
import sys
import random

import psychopy.visual
import psychopy.event
import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
rep_num = gui.data[1]

data_path = subj_id + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):

```

```

sys.exit("Data path " + data_path + " already exists!")

exp_data = []

bg_colour = [-1, -1, -0.25]

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30
oval_edge_contrast = 1.0
oval_line_width = 3.0
oval_n_edges = 128

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

oval = psychopy.visual.Circle(
    win=win,
    radius=oval_radius_pix,
    units="pix",
    edges=oval_n_edges,
    fillColor=oval_fill_colour,
)

vertices = oval.verticesPix.tolist()

right_vertices = vertices[: (oval_n_edges / 2 + 1)]

left_vertices = []

for vertex in right_vertices:
    left_vertices.append([-vertex[0], vertex[1]])

left_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=left_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[-oval_edge_contrast] * 3
)

right_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=right_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[+oval_edge_contrast] * 3
)

offsets = range(-180, 181, n_ovals_per_dim)

oval_line_contrasts = [0, 0.25, 0.5, 0.75, 1.0]
n_oval_line_contrasts = len(oval_line_contrasts)

n_trials_per_line_contrast = 10

trial_oval_line_contrasts = oval_line_contrasts * n_trials_per_line_contrast
random.shuffle(trial_oval_line_contrasts)

instructions = psychopy.visual.TextStim(
    win=win,
    wrapwidth=350,
)

instructions.text = """
Press a key from 1 to 5 to rate your sense of motion for the stimulus on a
given trial.\n
\n
Press any key to begin.
"""

```



```

instructions.draw()
win.flip()

psychoPy.event.waitKeys()

win.close()

```

6. What happens on a given trial?

On a given trial, we are going to show the stimulus with a given line contrast for 500ms, after a 500ms blank period, and then wait for a response from the participant, with a minimum inter-trial interval of 2 seconds. To draw the stimulus, we will follow the strategy we worked out in the demonstration code.

```

import os
import sys
import random

import psychoPy.visual
import psychoPy.event
import psychoPy.gui
import psychoPy.core

gui = psychoPy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
rep_num = gui.data[1]

data_path = subj_id + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

exp_data = []

bg_colour = [-1, -1, -0.25]

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30
oval_edge_contrast = 1.0
oval_line_width = 3.0
oval_n_edges = 128

win = psychoPy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

oval = psychoPy.visual.Circle(
    win=win,
    radius=oval_radius_pix,
    units="pix",
    edges=oval_n_edges,
    fillColor=oval_fill_colour,
)

vertices = oval.verticesPix.tolist()

right_vertices = vertices[::(oval_n_edges / 2 + 1)]

left_vertices = []

for vertex in right_vertices:
    left_vertices.append([-vertex[0], vertex[1]])

left_side = psychoPy.visual.ShapeStim(
    win=win,

```

```

    vertices=left_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[-oval_edge_contrast] * 3
)

right_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=right_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[+oval_edge_contrast] * 3
)

offsets = range(-180, 181, n_ovals_per_dim)

oval_line_contrasts = [0, 0.25, 0.5, 0.75, 1.0]
n_oval_line_contrasts = len(oval_line_contrasts)

n_trials_per_line_contrast = 10

trial_oval_line_contrasts = oval_line_contrasts * n_trials_per_line_contrast
random.shuffle(trial_oval_line_contrasts)

instructions = psychopy.visual.TextStim(
    win=win,
    wrapWidth=350,
)

instructions.text = """
Press a key from 1 to 5 to rate your sense of motion for the stimulus on a
given trial.\n
\n
Press any key to begin.
"""

instructions.draw()
win.flip()

psychopy.event.waitKeys()

pre_duration_s = 0.5
stim_duration_s = 0.5

min iti_s = 2.0

clock = psychopy.core.Clock()

for trial_oval_line_contrast in trial_oval_line_contrasts:

    left_side.lineColor = [-trial_oval_line_contrast] * 3
    right_side.lineColor = [+trial_oval_line_contrast] * 3

    clock.reset()

    while clock.getTime() < pre_duration_s:
        win.flip()

    while clock.getTime() < (pre_duration_s + stim_duration_s):

        row_count = 0

        for y_offset in offsets:

            ori = row_count * oval_row_ori_change

            for x_offset in offsets:

                ori = ori + oval_col_ori_change

            for stim in [oval, left_side, right_side]:
                stim.pos = [x_offset, y_offset]
                stim.ori = ori
                stim.draw()

            row_count = row_count + 1

```

```

win.flip()

win.flip()

keys = psychopy.event.waitKeys(
    keyList=["1", "2", "3", "4", "5", "q"],
    timeStamped=clock
)

while clock.getTime() < min_itit_s:
    win.flip()

win.close()

```

7. What data needs to be recorded for each trial?

For each trial, we need to record the line contrast and the response. While we are there, we will record the response time.

```

import os
import sys
import random

import psychopy.visual
import psychopy.event
import psychopy.gui
import psychopy.core

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
rep_num = gui.data[1]

data_path = subj_id + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

exp_data = []

bg_colour = [-1, -1, -0.25]

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30
oval_edge_contrast = 1.0
oval_line_width = 3.0
oval_n_edges = 128

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

oval = psychopy.visual.Circle(
    win=win,
    radius=oval_radius_pix,
    units="pix",
    edges=oval_n_edges,
    fillColor=oval_fill_colour,
)

vertices = oval.verticesPix.tolist()

right_vertices = vertices[: (oval_n_edges / 2 + 1)]

left_vertices = []

```

```

for vertex in right_vertices:
    left_vertices.append([-vertex[0], vertex[1]])

left_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=left_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[-oval_edge_contrast] * 3
)

right_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=right_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[+oval_edge_contrast] * 3
)

offsets = range(-180, 181, n_ovals_per_dim)

oval_line_contrasts = [0, 0.25, 0.5, 0.75, 1.0]
n_oval_line_contrasts = len(oval_line_contrasts)

n_trials_per_line_contrast = 10

trial_oval_line_contrasts = oval_line_contrasts * n_trials_per_line_contrast
random.shuffle(trial_oval_line_contrasts)

instructions = psychopy.visual.TextStim(
    win=win,
    wrapWidth=350,
)

instructions.text = """
Press a key from 1 to 5 to rate your sense of motion for the stimulus on a
given trial.\n
\n
Press any key to begin.
"""

instructions.draw()
win.flip()

psychopy.event.waitKeys()

pre_duration_s = 0.5
stim_duration_s = 0.5

min_iti_s = 2.0

clock = psychopy.core.Clock()

for trial_oval_line_contrast in trial_oval_line_contrasts:

    left_side.lineColor = [-trial_oval_line_contrast] * 3
    right_side.lineColor = [+trial_oval_line_contrast] * 3

    clock.reset()

    while clock.getTime() < pre_duration_s:
        win.flip()

    while clock.getTime() < (pre_duration_s + stim_duration_s):

        row_count = 0

        for y_offset in offsets:

            ori = row_count * oval_row_ori_change

            for x_offset in offsets:

                ori = ori + oval_col_ori_change

            for stim in [oval, left_side, right_side]:

```

```

        stim.pos = [x_offset, y_offset]
        stim.ori = ori
        stim.draw()

        row_count = row_count + 1

    win.flip()

win.flip()

keys = psychopy.event.waitKeys(
    keyList=["1", "2", "3", "4", "5", "q"],
    timeStamped=clock
)

for key in keys:
    if key[0] == "q":
        sys.exit("User quit")

    key_num = int(key[0])
    rt = key[1]

    trial_data = [trial_oval_line_contrast, key_num, rt]

    exp_data.append(trial_data)

    while clock.getTime() < min_iti_s:
        win.flip()

win.close()

```

Notice that we have converted the numeric responses, which are received as strings, into integers. We have also allowed a "q" response, which terminates the program. This is often a good thing to add to your program, at least during development.

8. How should the data be saved for further analysis?

We want to save the data as a plain-text file, where each row is a trial and the columns are the line contrast, the rating, and the response time.

```

import os
import sys
import random

import numpy as np

import psychopy.visual
import psychopy.event
import psychopy.gui
import psychopy.core

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Repeat num:")

gui.show()

subj_id = gui.data[0]
rep_num = gui.data[1]

data_path = subj_id + "_rep_" + rep_num + ".tsv"

if os.path.exists(data_path):
    sys.exit("Data path " + data_path + " already exists!")

exp_data = []

bg_colour = [-1, -1, -0.25]

oval_radius_pix = [10, 18]
oval_fill_colour = [-1, 0.25, -1]
n_ovals_per_dim = 40
oval_col_ori_change = -30
oval_row_ori_change = +30

```

```

oval_edge_contrast = 1.0
oval_line_width = 3.0
oval_n_edges = 128

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=bg_colour
)

oval = psychopy.visual.Circle(
    win=win,
    radius=oval_radius_pix,
    units="pix",
    edges=oval_n_edges,
    fillColor=oval_fill_colour,
)

vertices = oval.verticesPix.tolist()

right_vertices = vertices[: (oval_n_edges / 2 + 1)]

left_vertices = []

for vertex in right_vertices:
    left_vertices.append([-vertex[0], vertex[1]])

left_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=left_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[-oval_edge_contrast] * 3
)

right_side = psychopy.visual.ShapeStim(
    win=win,
    vertices=right_vertices,
    closeShape=False,
    units="pix",
    lineWidth=oval_line_width,
    lineColor=[+oval_edge_contrast] * 3
)

offsets = range(-180, 181, n_ovals_per_dim)

oval_line_contrasts = [0, 0.25, 0.5, 0.75, 1.0]
n_oval_line_contrasts = len(oval_line_contrasts)

n_trials_per_line_contrast = 10

trial_oval_line_contrasts = oval_line_contrasts * n_trials_per_line_contrast
random.shuffle(trial_oval_line_contrasts)

instructions = psychopy.visual.TextStim(
    win=win,
    wrapWidth=350,
)

instructions.text = """
Press a key from 1 to 5 to rate your sense of motion for the stimulus on a
given trial.\n
\n
Press any key to begin.
"""

instructions.draw()
win.flip()

psychopy.event.waitKeys()

pre_duration_s = 0.5
stim_duration_s = 0.5

min_iti_s = 2.0

clock = psychopy.core.Clock()

```

```

for trial_oval_line_contrast in trial_oval_line_contrasts:

    left_side.lineColor = [-trial_oval_line_contrast] * 3
    right_side.lineColor = [+trial_oval_line_contrast] * 3

    clock.reset()

    while clock.getTime() < pre_duration_s:
        win.flip()

    while clock.getTime() < (pre_duration_s + stim_duration_s):

        row_count = 0

        for y_offset in offsets:

            ori = row_count * oval_row_ori_change

            for x_offset in offsets:

                ori = ori + oval_col_ori_change

                for stim in [oval, left_side, right_side]:
                    stim.pos = [x_offset, y_offset]
                    stim.ori = ori
                    stim.draw()

                row_count = row_count + 1

            win.flip()

        win.flip()

    keys = psychopy.event.waitKeys(
        keyList=["1", "2", "3", "4", "5", "q"],
        timeStamped=clock
    )

    for key in keys:

        if key[0] == "q":
            sys.exit("User quit")

        key_num = int(key[0])
        rt = key[1]

    trial_data = [trial_oval_line_contrast, key_num, rt]

    exp_data.append(trial_data)

    while clock.getTime() < min_iti_s:
        win.flip()

np.savetxt(data_path, exp_data, delimiter="\t")

win.close()

```

Summary

Here, we have been able to put together another full example experiment. The most complicated aspect was the stimulus creation, but we ended up being able to combine some simple psychopy components to generate quite a complex stimulus. We also used a randomised trial sequence and a different kind of task.

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. **Exercise solutions**
 14. [Extra: Spatial frequency filtering](#)
-

Exercise solutions

Scroll down to see the solutions to the exercises. The different lessons have quite large spacing so that you don't accidentally see the solutions for other lessons.

Getting started

1.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

text = psychopy.visual.TextStim(
    win=win,
    text="Hello, world!",
    color=[-1, 0, -1]
)

text.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```

2. The `flipHoriz` argument, when set to `True`, causes the text to be drawn "flipped" horizontally. This can be useful when presenting stimuli through a mirror, such as when showing different stimuli to the two eyes using a stereoscope or when showing stimuli in an fMRI scanner.
3. Specifying dimensions in "deg" requires knowledge of the viewing distance and the stimulus size, in the same units.

Drawing to a window

1.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

text = psychopy.visual.TextStim(
    win=win,
    text="Hello, world!",
    color=[-1, -1, -1]
)

for offset in range(20):

    text.pos = [offset, offset]

    text.draw()

text.color = [-1, 0, -1]
text.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```

2.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

text = psychopy.visual.TextStim(
    win=win,
    text="Hello, world!",
    color=[-1, -1, -1]
)

for h_offset in [-120, 0, 120]:

    for v_offset in range(-150, 155, 20):

        text.pos = [h_offset, v_offset]

        text.color = [-1, -1, -1]

        mod_intensity = v_offset / 150.

        if h_offset == -120:
            text.color[0] = mod_intensity
        elif h_offset == 0:
            text.color[1] = mod_intensity
        elif h_offset == 120:
            text.color[2] = mod_intensity

        text.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```

Drawing—gratings

1. We can see what the raised cosine mask looks like by setting the `mask` parameter to `"raisedCos"`:

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
    size=[150, 150]
)

grating.mask = "raisedCos"

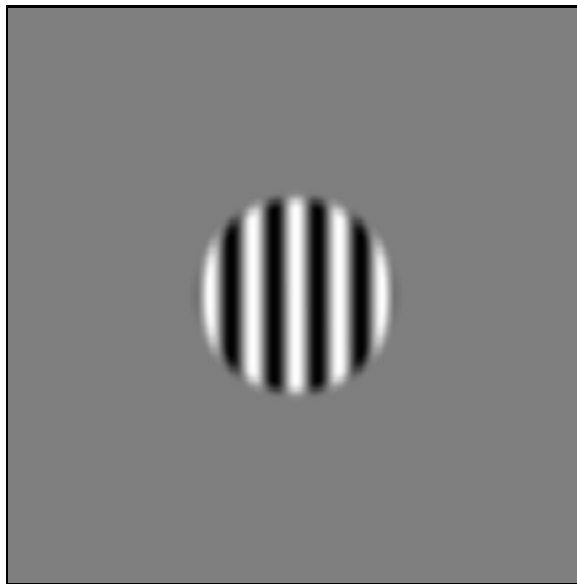
grating.sf = 5.0 / 150.0

grating.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



We can see that this mask is somewhat of a compromise between a circle and a Gaussian mask. We avoid the sharp edges of the circle, while having more of the stimulus visible than is the case with a Gaussian mask. This is frequently a useful mask in situations where you would like participants to see an extended stimulus but without a sharp border.

- 2.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
    size=[150, 150],
    mask="circle",
    sf=5.0 / 150.0
)

grating.draw()
```

```

grating.pos = [150 * 0.75, 150 * 0.75]

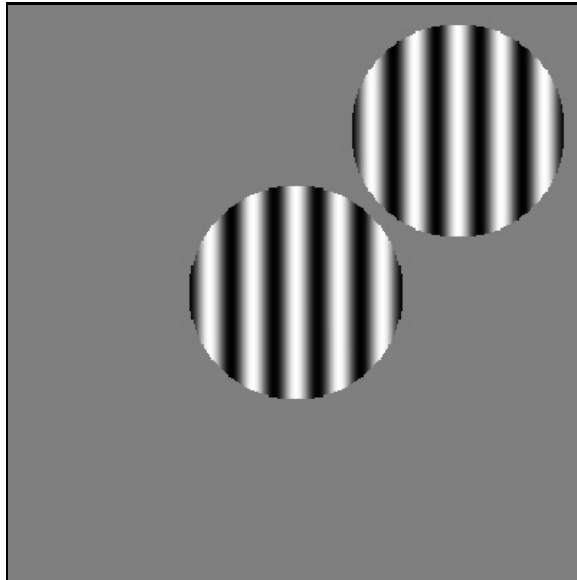
grating.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

```



As seen in the above, the "grey" regions outside the circular mask are not actually grey but are transparent. The "square" of the second grating is able to encroach on the first grating without overwriting parts of it in grey.

3.

```

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    units="pix",
    size=[180, 180]
)

grating.mask = "circle"
grating_hpos = [-100, 100]
grating.sf = 5.0 / 180.0

# first, background gratings
bg_oris = [0.0, 90.0]
grating.contrast = 0.5

for i_grating in range(2):

    grating.pos = [grating_hpos[i_grating], 0]
    grating.ori = bg_oris[i_grating]

    grating.draw()

# now, the foreground gratings
grating.size = [80, 80]
grating.ori = 0.0
grating.contrast = 0.2

for i_grating in range(2):

    grating.pos = [grating_hpos[i_grating], 0]

    grating.draw()

win.flip()

```

```
psychopy.event.waitKeys()
```

```
win.close()
```

Drawing—shapes

1.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

circle = psychopy.visual.Circle(
    win=win,
    units="pix",
    radius=50,
    fillColor=[-1] * 3,
    lineColor=[-1] * 3,
    edges=128
)

for h_off in [-1, +1]:
    for v_off in [-1, +1]:

        circle.pos = [100 * h_off, 100 * v_off]

        circle.draw()

rect = psychopy.visual.Rect(
    win=win,
    units="pix",
    width=200,
    height=200,
    lineColor=[1] * 3,
    fillColor=[1] * 3
)

rect.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```

2.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False,
    color=[1, 1, 1]
)

circle = psychopy.visual.Circle(
    win=win,
    units="pix",
    radius=50,
    fillColor=[-1] * 3,
    lineColor=[-1] * 3,
    edges=128
)

for h_off in [-1, +1]:
    for v_off in [-1, +1]:

        circle.pos = [100 * h_off, 100 * v_off]

        circle.draw()

rect = psychopy.visual.Rect(
    win=win,
    units="pix",
    width=200,
)


```

```
rect.draw()
```

Drawing—images

1.

2.

```
import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

grating = psychopy.visual.GratingStim(
    win=win,
    size=[200, 200],
    units="pix",
    sf=5.0 / 200.0,
    mask="circle"
)

for grating_ori in [0, 90]:
    grating.ori = grating_ori

    grating.opacity = 0.5

    grating.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```


Drawing—dots

1.

```
import random

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    units="pix",
    fullscr=False
)

offsets = range(-180, 181, 20)

base_ori = 0.0
odd_ori = 45.0

dot_xys = []
dot_oris = []
dot_phases = []

for h_offset in offsets:
    for v_offset in offsets:

        dot_xys.append([h_offset, v_offset])
        dot_oris.append(base_ori)
        dot_phases.append(random.uniform(0, 1))

n_dots = len(dot_xys)

i_mod_dot = random.choice(range(n_dots))

dot_oris[i_mod_dot] = odd_ori

dot_stim = psychopy.visual.ElementArrayStim(
    win=win,
    units="pix",
    nElements=n_dots,
    elementTex="sin",
    elementMask="gauss",
    sfs=5.0 / 2.5,
    xys=dot_xys,
    sizes=20,
    oris=dot_oris,
    phases=dot_phases
)

dot_stim.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```

Collecting responses

1. When pressed on the horizontal row of numeric keys, the value returned is a string representation of the number. When the same number is pressed on the numpad, the string is prepended with `num_`—for example, `num_1`. This is true when "numlock" is enabled.
2. No, caps lock appears to have little effect on the returned string. For example, having caps lock turned on and pressing the "d" key returns `d`, not `D`.

Providing input

1.

```
import psychopy.gui

gui = psychopy.gui.Dlg()

gui.addField("Subject ID:")
gui.addField("Condition Num:")

gui.show()

subj_id = gui.data[0]
cond_num = gui.data[1]

data_path = subj_id + "_experiment_cond_" + cond_num + ".tsv"
```

Saving data

1. The `header` argument accepts a string, which is saved at the top of the file.

```
import random
import numpy as np
import pprint

data = []

for trial in range(10):

    data.append(
        [
            random.uniform(0, 180),
            random.choice([1, 2])
        ]
    )

pprint.pprint(data)

header = "In the second column below, 1 means left and 2 means right"

np.savetxt(
    "p1000_exp_cond_1_run_2.tsv",
    data,
    delimiter="\t",
    header=header
)
```

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Vision Science

0. [Introduction](#)
 1. [Getting started](#)
 2. [Drawing to a window](#)
 3. [Drawing—gratings](#)
 4. [Drawing—shapes](#)
 5. [Drawing—images](#)
 6. [Drawing—dots](#)
 7. [Temporal dynamics](#)
 8. [Collecting responses](#)
 9. [Providing input](#)
 10. [Saving data](#)
 11. [Putting it all together—a framework and an example](#)
 12. [Putting it all together—another example](#)
 13. [Exercise solutions](#)
 14. **Extra: Spatial frequency filtering**
-

Extra: Spatial frequency filtering

Objectives

- Be able to convert stimuli from image space to frequency space.
- Know how to create spatial frequency filters.
- Be able to apply spatial frequency filters to produce filtered images.

Many groups are interested in using filtering techniques to alter the spatial frequency content of their stimuli. Here, we will learn some of the ways that we can filter images in Python.

N.B. We touch on some advanced concepts in this lesson. You might find it useful to read [my lesson on arrays](#) (from another course) to understand our data representations.

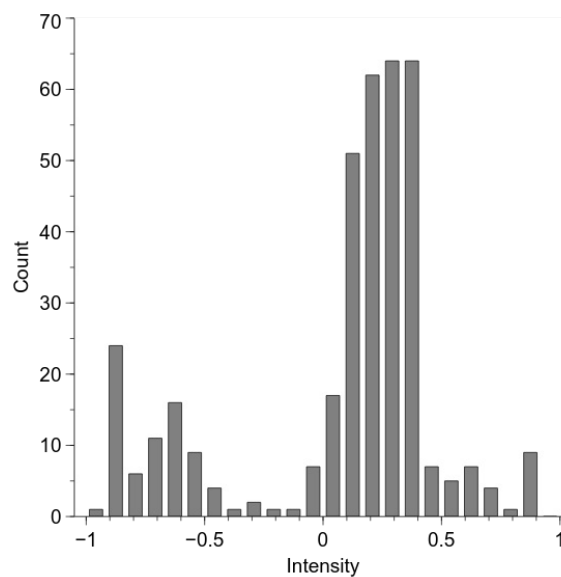
Preparing the stimulus

For this lesson, we are going to use the [image of UNSW](#) shown below:

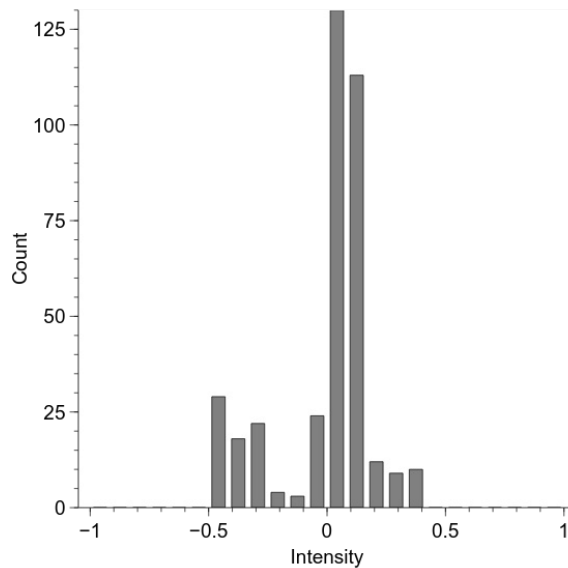


First, however, we need to think about the intensity structure of the image. Because we will be altering such structure during the filtering operations, we want to ensure that we a) preserve important properties of the image, and b) allow sufficient 'room' so that we don't run into floor and ceiling issues (that is, intensities that are outside of the range that we can display).

Let's start by having a look at the intensity histogram of the image. That is, we will count up how many pixels have a particular intensity value for each of a range of intensities between -1 and 1. We get something like below:



We will now adjust the histogram to make it be able to be preserved across our spatial filtering. We will do this by first making the mean intensity to be zero and then alter the standard deviation to be 0.2. This standard deviation is a measure of image contrast, also known as "root mean square" (RMS) contrast. We will see how to do this in code and what effect it has on image appearance below, but here is the effect on the intensity histogram:



Our aim will be to preserve the mean and standard deviation of this intensity distribution for each of our filtered images.

Displaying altered images in PsychoPy

We have considered previously how to [use images from a file on disk](#). Here, we wish to do something similar but also slightly different—we want to use an image from a file on disk, but we want to first load it into Python so we can make some alterations to it.

```
import numpy as np
import scipy.misc

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    fullscr=False,
    units="pix"
)

# this gives a (y, x) array of values between 0.0 and 255.0
raw_img = scipy.misc.imread(
    "unsw_bw.jpg",
    flatten=True
)

# we first need to convert it to the -1:+1 range
raw_img = (raw_img / 255.0) * 2.0 - 1.0

# we also need to flip it upside-down to match the psychopy convention
raw_img = np.flipud(raw_img)

img = psychopy.visual.ImageStim(
    win=win,
    image=raw_img,
    units="pix",
    size=raw_img.shape
)

img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```



As seen in the above, we first use the function `scipy.misc.imread` to load the image into the variable `raw_img`. After converting it into the -1 to +1 range, we flip it upside-down and then provide `ImageStim` with the variable `raw_img`, which contains the direct image data rather than the filename that we have used previously.

By doing it this way, we can alter the image—such as performing the histogram changes we looked at previously:

```
import numpy as np
import scipy.misc

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    fullscr=False,
    units="pix"
)

# this gives a (y, x) array of values between 0.0 and 255.0
raw_img = scipy.misc.imread(
    "unsw_bw.jpg",
    flatten=True
)

# we first need to convert it to the -1:+1 range
raw_img = (raw_img / 255.0) * 2.0 - 1.0

# we also need to flip it upside-down to match the psychopy convention
raw_img = np.flipud(raw_img)

# desired RMS
rms = 0.2

# make the mean to be zero
raw_img = raw_img - np.mean(raw_img)
# make the standard deviation to be 1
raw_img = raw_img / np.std(raw_img)
# make the standard deviation to be the desired RMS
raw_img = raw_img * rms

img = psychopy.visual.ImageStim(
    win=win,
    image=raw_img,
    units="pix",
    size=raw_img.shape
)

img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()
```




Converting to frequency space

We are going to perform spatial frequency filtering via the frequency domain. That is, we are going to convert our image representation from horizontal and vertical space to a polar representation of orientation (polar angle) and spatial frequency (radius). We are not altering anything about the image in this transformation; we are simply converting it into a more convenient format in which to perform our filtering.

In the below code, we use the `fft2` function (Fast Fourier Transform) to convert our image. We then use the `abs` function to get the amplitude spectrum, and use `fftshift` to move the origin to the centre of the image. Because many of the values are quite small, we use a log transform to make the spectrum easier to see (adding a small value to prevent logs of zero).

```
import numpy as np
import scipy.misc

import psychopy.visual
import psychopy.event

win = psychopy.visual.Window(
    size=[400, 400],
    fullscr=False,
    units="pix"
)

# this gives a (y, x) array of values between 0.0 and 255.0
raw_img = scipy.misc.imread(
    "unsw_bw.jpg",
    flatten=True
)

# we first need to convert it to the -1:+1 range
raw_img = (raw_img / 255.0) * 2.0 - 1.0

# we also need to flip it upside-down to match the psychopy convention
raw_img = np.flipud(raw_img)

# desired RMS
rms = 0.2

# make the mean to be zero
raw_img = raw_img - np.mean(raw_img)
# make the standard deviation to be 1
raw_img = raw_img / np.std(raw_img)
# make the standard deviation to be the desired RMS
raw_img = raw_img * rms

# convert to frequency domain
img_freq = np.fft.fft2(raw_img)

# calculate amplitude spectrum
img_amp = np.fft.fftshift(np.abs(img_freq))

# for display, take the Logarithm
```

```

img_amp_disp = np.log(img_amp + 0.0001)

# rescale to -1:+1 for display
img_amp_disp = (
    (img_amp_disp - np.min(img_amp_disp)) * 2
) / np.ptp(img_amp_disp) # 'ptp' = range
) - 1

img = psychopy.visual.ImageStim(
    win=win,
    image=img_amp_disp,
    units="pix",
    size=raw_img.shape
)

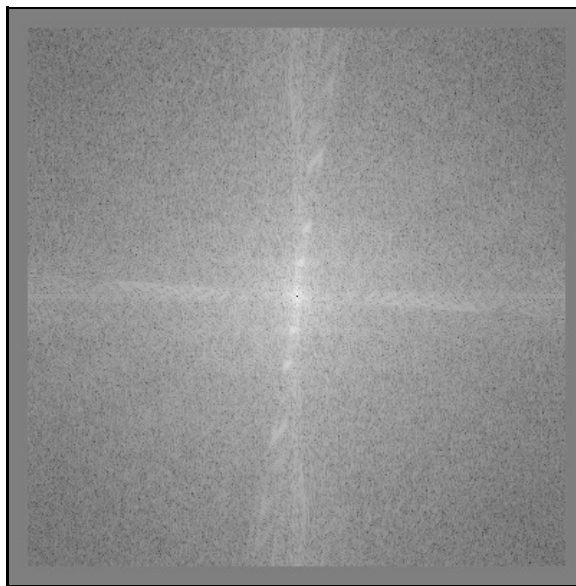
img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

```



What we will do in our spatial frequency filtering is alter the radial structure of this frequency-space representation, and then convert it back into the image domain.

Creating a spatial frequency filter

Here, we will look at how to construct an appropriate spatial frequency filter. We will use the Butterworth class of filters, beginning with a low-pass filter. To create such a filter, we first need to decide on two parameters—the cutoff frequency and the filter 'order'. The cutoff frequency is typically between 0 and 0.5, and determine the distance from the origin at which the filter response is at half its maximum. The 'order' is an integer that determines the steepness of the filter about this value, with higher values giving steeper responses.

We can create a low-pass Butterworth filter in Python using the `psychopy.filters.butter2d_lp` function. Let's see what one looks like:

```

import numpy as np
import scipy.misc

import psychopy.visual
import psychopy.event
import psychopy.filters

win = psychopy.visual.Window(
    size=[400, 400],
    fullscr=False,
    units="pix"
)

# this gives a (y, x) array of values between 0.0 and 255.0
raw_img = scipy.misc.imread(

```

```

"unsw_bw.jpg",
  flatten=True
)

# we first need to convert it to the -1:+1 range
raw_img = (raw_img / 255.0) * 2.0 - 1.0

# we also need to flip it upside-down to match the psychopy convention
raw_img = np.flipud(raw_img)

# desired RMS
rms = 0.2

# make the mean to be zero
raw_img = raw_img - np.mean(raw_img)
# make the standard deviation to be 1
raw_img = raw_img / np.std(raw_img)
# make the standard deviation to be the desired RMS
raw_img = raw_img * rms

# convert to frequency domain
img_freq = np.fft.fft2(raw_img)

# calculate amplitude spectrum
img_amp = np.fft.fftshift(np.abs(img_freq))

lp_filt = psychopy.filters.butter2d_lp(
  size=raw_img.shape,
  cutoff=0.05,
  n=10
)

# 'lp_filt' will be 0:1; convert it to -1:1 for display
lp_filt_disp = lp_filt * 2.0 - 1.0

img = psychopy.visual.ImageStim(
  win=win,
  image=lp_filt_disp,
  units="pix",
  size=raw_img.shape
)

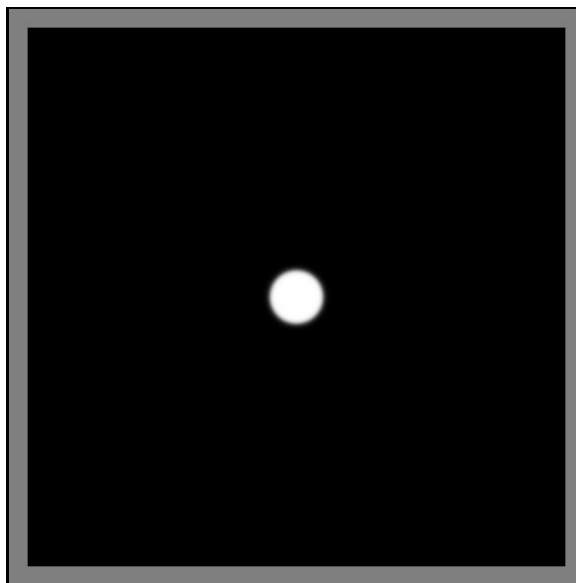
img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

```



Applying a spatial frequency filter

To apply our filter, we simply multiply the frequency-space representation of our image by the filter shown above:

```

import numpy as np
import scipy.misc

import psychopy.visual
import psychopy.event
import psychopy.filters

win = psychopy.visual.Window(
    size=[400, 400],
    fullscr=False,
    units="pix"
)

# this gives a (y, x) array of values between 0.0 and 255.0
raw_img = scipy.misc.imread(
    "unsw_bw.jpg",
    flatten=True
)

# we first need to convert it to the -1:+1 range
raw_img = (raw_img / 255.0) * 2.0 - 1.0

# we also need to flip it upside-down to match the psychopy convention
raw_img = np.flipud(raw_img)

# desired RMS
rms = 0.2

# make the mean to be zero
raw_img = raw_img - np.mean(raw_img)
# make the standard deviation to be 1
raw_img = raw_img / np.std(raw_img)
# make the standard deviation to be the desired RMS
raw_img = raw_img * rms

# convert to frequency domain
img_freq = np.fft.fft2(raw_img)

# calculate amplitude spectrum
img_amp = np.fft.fftshift(np.abs(img_freq))

lp_filt = psychopy.filters.butter2d_lp(
    size=raw_img.shape,
    cutoff=0.05,
    n=10
)

img_filt = np.fft.fftshift(img_freq) * lp_filt

img_filt_amp = np.abs(img_filt)

# for display, take the Logarithm
img_filt_amp_disp = np.log(img_filt_amp + 0.0001)

# rescale to -1:+1 for display
img_filt_amp_disp = (
    (img_filt_amp_disp - np.min(img_filt_amp_disp)) * 2
) / np.ptp(img_filt_amp_disp) # 'ptp' = range
) - 1

img = psychopy.visual.ImageStim(
    win=win,
    image=img_filt_amp_disp,
    units="pix",
    size=raw_img.shape
)

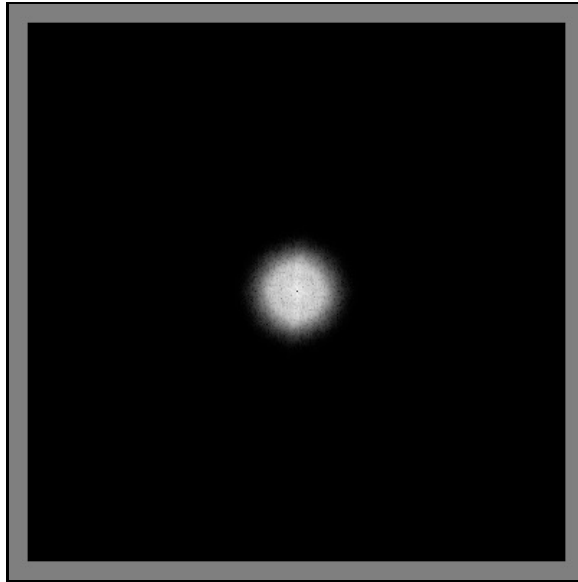
img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

```



Converting back to an image

Now that we have our altered frequency-space representation, we convert it back to image-space by performing the inverse of the operations we performed to get it into frequency-space.

```
import numpy as np
import scipy.misc

import psychopy.visual
import psychopy.event
import psychopy.filters

win = psychopy.visual.Window(
    size=[400, 400],
    fullscr=False,
    units="pix"
)

# this gives a (y, x) array of values between 0.0 and 255.0
raw_img = scipy.misc.imread(
    "unsw_bw.jpg",
    flatten=True
)

# we first need to convert it to the -1:+1 range
raw_img = (raw_img / 255.0) * 2.0 - 1.0

# we also need to flip it upside-down to match the psychopy convention
raw_img = np.flipud(raw_img)

# desired RMS
rms = 0.2

# make the mean to be zero
raw_img = raw_img - np.mean(raw_img)
# make the standard deviation to be 1
raw_img = raw_img / np.std(raw_img)
# make the standard deviation to be the desired RMS
raw_img = raw_img * rms

# convert to frequency domain
img_freq = np.fft.fft2(raw_img)

# calculate amplitude spectrum
img_amp = np.fft.fftshift(np.abs(img_freq))

lp_filt = psychopy.filters.butter2d_lp(
    size=raw_img.shape,
    cutoff=0.05,
    n=10
)

img_filt = np.fft.fftshift(img_freq) * lp_filt
```

```

# convert back to an image
img_new = np.real(np.fft.ifft2(np.fft.ifftshift(img_filt)))

# convert to mean zero and specified RMS contrast
img_new = img_new - np.mean(img_new)
img_new = img_new / np.std(img_new)
img_new = img_new * rms

# there may be some stray values outside of the presentable range; convert < -1
# to -1 and > 1 to 1
img_new = np.clip(img_new, a_min=-1.0, a_max=1.0)

img = psychopy.visual.ImageStim(
    win=win,
    image=img_new,
    units="pix",
    size=raw_img.shape
)

img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

```



And we have our low-pass filtered image!

Other spatial frequency filters

The above process was for a low-pass filter, but similar strategies can be adopted for high-pass and band-pass filters. For a high-pass filter, you can use `psychopy.filters.butter2d_hp`, which has similar arguments as the low-pass filter. For a band-pass filter, you can use `psychopy.filters.butter2d_bp`, which requires separate cutoff frequencies for the inner and outer frequencies that define the inclusive frequency band.

Here is an example of a high-pass filter in action:

```

import numpy as np
import scipy.misc

import psychopy.visual
import psychopy.event
import psychopy.filters

win = psychopy.visual.Window(
    size=[400, 400],
    fullscr=False,
    units="pix"
)

# this gives a (y, x) array of values between 0.0 and 255.0
raw_img = scipy.misc.imread(

```

```

    "unsw_bw.jpg",
    flatten=True
)

# we first need to convert it to the -1:+1 range
raw_img = (raw_img / 255.0) * 2.0 - 1.0

# we also need to flip it upside-down to match the psychopy convention
raw_img = np.flipud(raw_img)

# desired RMS
rms = 0.2

# make the mean to be zero
raw_img = raw_img - np.mean(raw_img)
# make the standard deviation to be 1
raw_img = raw_img / np.std(raw_img)
# make the standard deviation to be the desired RMS
raw_img = raw_img * rms

# convert to frequency domain
img_freq = np.fft.fft2(raw_img)

# calculate amplitude spectrum
img_amp = np.fft.fftshift(np.abs(img_freq))

hp_filt = psychopy.filters.butter2d_hp(
    size=raw_img.shape,
    cutoff=0.05,
    n=10
)

img_filt = np.fft.fftshift(img_freq) * hp_filt

# convert back to an image
img_new = np.real(np.fft.ifft2(np.fft.ifftshift(img_filt)))

# convert to mean zero and specified RMS contrast
img_new = img_new - np.mean(img_new)
img_new = img_new / np.std(img_new)
img_new = img_new * rms

# there may be some stray values outside of the presentable range; convert < -1
# to -1 and > 1 to 1
img_new = np.clip(img_new, a_min=-1.0, a_max=1.0)

img = psychopy.visual.ImageStim(
    win=win,
    image=img_new,
    units="pix",
    size=raw_img.shape
)

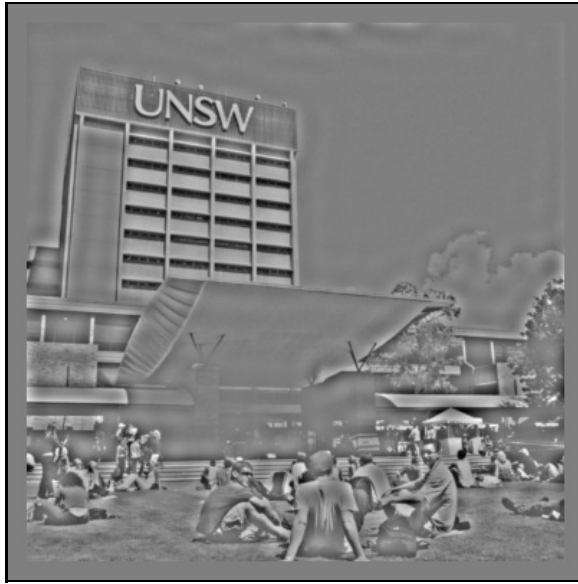
img.draw()

win.flip()

psychopy.event.waitKeys()

win.close()

```



[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)