Getting started with Python and PyLink

SR Research Ltd., Ottawa, Canada

Last updated: September 26, 2024

1 Choose a library to build your experimental tasks

A computer-based experimental task is essentially a “video game”. Visual and auditory stimuli are presented, and the participants respond to various task manipulations with a keyboard, a mouse, or a gamepad. In principle, one can use any multimedia library to program an experimental task, though some libraries may not be suitable for tasks that require precise timing. There are many multimedia libraries that one can use to program experimental tasks in Python, such as pygame and pyglet.

Some libraries are explicitly built for psychology experiments and included primarily as an integrated part of complete application suites (for example, PsychoPy, expyriment, Vision EGG, and OpenSesame). Those application suites are a popular choice among many Python users as they are feature-enriched with carefully implemented methods for experimental purposes, for example, a staircase procedure or complex visual stimuli (e.g., checkerboards and gratings). The graphical interface for experimental design provided by those application suites is a leading reason many users choose to use these applications because an intimate knowledge of EyeLink integration conventions via PyLink scripting is not part of their typical use.

Although this document is otherwise primarily intended for users who do wish to design their experiments by means of writing their own scripted code, there is nonetheless a degree of overlap in the utility of information below and some users of the aforementioned application suites. For example, PsychoPy users who use its Coder interface, or those who use its Builder interface and also choose to adapt an element of Python scripting using Code Components can still benefit from some of the information below.

The EyeLink Developers Kit comes with various sets of example projects demonstrating the various ways EyeLink integration can be accomplished using some of these application suites. More information relating to using these applications for EyeLink experiments can be found on our support forums, here:

- Getting Started with PsychoPy / EyeLink Integration
- Getting Started with OpenSesame

Otherwise the following information in this document is more generalized, and covers the core principles for using PyLink, from installation through to implementation of EyeLink integrations within the context of basic Python environments and extensions thereof including virtual environments such as those using *venv* or Conda. In this capacity, the PyLink integrations discussed below largely use the pygame library (its use, like PyLink is also agnostic to the Python-environment in which it is used) as an exemplary multimedia library as the tool through which PyLink effects its audiovisual presentation for experimental display.

pygame is an excellent choice in this regard providing a lightweight solution for many research projects. The EyeLink Developers Kit also comes with a set of examples demonstrating pygame integration as well, which will be further elaborated in detail below in this document. You will also find examples for other Python-based task creation tools, such as OpenSesame, on the SR Research Support Forum (https://www.sr-research.com/support/thread-52.html).

2 Install Python and Python modules

Python is a popular high-level programming language in scientific computation. There are quite a few distributions (e.g., Anaconda, Canopy), which bundle features and packages that are not part of the official Python distribution. This guide will assume that you would use the official Python distribution that is freely available from https://www.python.org, or you are using the Standalone version of PsychoPy, which bundles a copy of Python. As support for Python 2 has ended, here we assume you are using Python 3 instead.

This short guide will not cover the syntax and building blocks of the Python language. For beginners, the official Python tutorial is what we would recommend (https://docs.python.org/3/tutorial/).

2.1 Install Python

Your Mac may ship with Python 3 pre-installed by default. To check if Python has been installed, launch a terminal and type *python3* at the command line prompt (see the output below). If you do not have Python 3 on your Mac, please download the installer from https://www.python.org/downloads/mac-osx/ and install it. Note with version 2.1.762 or later of EyeLink Developers Kit, the PyLink library will work with versions of Python installed with the Intel Installer or the universal2 installer for Macs.

```
eyelink@eyelinks-MacBook-Pro ~ % python3
Python 3.7.9 (v3.7.9:13c94747c7, Aug 15 2020, 01:31:08)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you use Ubuntu or another Debian-based Linux distribution, run the following command in a terminal to install Python 3.6 or a later version (Ubuntu now bundles with Python 3 by default).

```
sudo apt-get install python3.6
```

To install Python 3 on a Windows PC, please download the relevant installer (https://www.python.org/downloads/windows/).

2.2 Install Python modules

Thousands of Python modules (libraries) are actively developed and maintained by the community. The preferred installer for Python modules is *pip*, which has been part of the official Python distribution since version 2.7.9. A complete guide on pip can be found on Python.org, https://docs.python.org/3/installing/index.html. Here we will give a very brief discussion on *pip*, with example commands.

2.2.1 Install Python modules on macOS / Linux

On macOS and Linux systems, the *pip install* command takes the following format.

```
python3 -m pip install SomePackage --user
```

The above command will install a (or multiple) module(s) for the default Python 3 version on your computer. The above example uses the *–user* flag for installation into the current user's HOME directory path where Python libraries can be installed without root access. This installation method is most commonly used and secured against making system-wide changes. For more information the related section of the official *pip User Guide* can be further consulted, here: https://pip.pypa.io/en/stable/user_guide/#user-installs.

In other installation scenarios (e.g., to obtain administrator privilege on macOS and Linux), you may need to add *sudo* before the command if your desire is to perform the installation into Python's system-wide packages available for all users. If you are installing modules for a particular version of Python (e.g., when you have multiple versions installed on your computer), the recommended command takes the following format. The *python3* command now includes an extra version number (e.g., *python3.7*, *python3.9*).

```
python3.x -m pip install SomePackage --user
```

The *pip* command, by default, will install the latest version of a module. If you would like to install a particular version of a module you can include the module version number in the command. For instance, the example command below will install pygame 1.9.6 instead of the latest version (2.x).

```
eyelink@eyelinks-Mac-mini-3 ~ % python3.6 -m pp install pygame==1.9.6
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6: No module named pp
eyelink@eyelinks-Mac-mini-3 ~ % python3.6 -m pip install pygame==1.9.6
Collecting pygame==1.9.6
  Downloading pygame-1.9.6-cp36-cp36m-macosx_10_11_intel.whl (4.9 MB)
     |████████████████████████████████| 4.9 MB 109 kB/s
Installing collected packages: pygame
Successfully installed pygame-1.9.6
```

It is important to upgrade your pip to the latest version (see the command below) so that it recognizes *universal2* tagged wheel files on macOS and *manylinux* tagged wheel files on Linux as they are used in the PyLink packages bundled with the version 2.1.762 or later of EyeLink Developers Kit and/or those published to pypi.org. The *universal2* wheel files support both Intel x86_64 (which loads on Intel machines, and on Apple Silicon when running non-*universal2* Intel Python versions under Rosetta 2 emulation) and arm64 (which is native for macOS on Apple Silicon) architectures.

```
python3.6 -m pip install --upgrade pip --user
```

Note the *--upgrade pip* option doesn't work with earlier version of Python 2.7. If you are using an old version, first download ***https://bootstrap.pypa.io/pip/2.7/get-pip.py***, then run *python2.7 get-pip.py* from shell to install the latest pip version for Python 2.7.

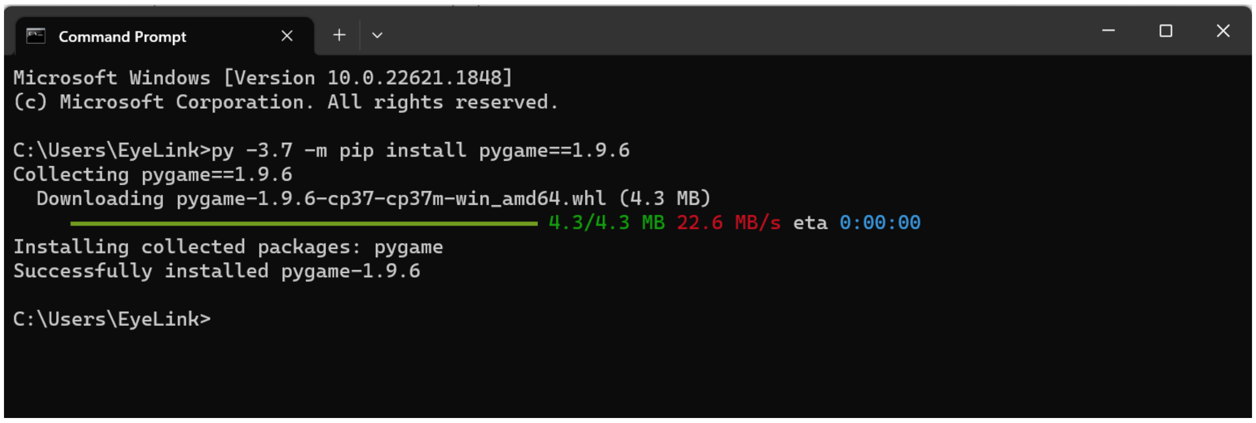
2.2.2 Install Python modules on Windows

On Windows, the *py* launcher is now the recommended method for launching Python. So, *py -3* will launch the default Python 3 on your system, and *py -3.8* will launch Python version 3.8. Consequently, the *pip* command will take the following format.

```
py -3.x -m pip install SomePackage
```

2.2.3 Install pygame

To install pygame, simply run the *pip* command matching the recommended syntax above. For example, the command shown in the screenshot below installs pygame (version 1.9.6) for Python 3.7 on a Windows 11 machine.



```
Microsoft Windows [Version 10.0.22621.1848]
(c) Microsoft Corporation. All rights reserved.

C:\Users\EyeLink>py -3.7 -m pip install pygame==1.9.6
Collecting pygame==1.9.6
  Downloading pygame-1.9.6-cp37-cp37m-win_amd64.whl (4.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.3/4.3 MB 22.6 MB/s eta 0:00:00
Installing collected packages: pygame
Successfully installed pygame-1.9.6

C:\Users\EyeLink>
```

For more general information about the pygame project, see their Getting Started Wiki.

3 Install the EyeLink Developers Kit and PyLink

3.1 Install the EyeLink Developers Kit

EyeLink eye trackers come with an application programming interface (API, included in the EyeLink Developers Kit), which allows users to develop custom eye tracking applications. The Python wrapper of this API is the PyLink library, which is compatible with both Python 2 and 3.

To use PyLink, you need first to install the latest version of the EyeLink Developers Kit, which works on all major platforms (Windows, macOS, and Ubuntu). Windows and macOS installers for the EyeLink Developers Kit are freely available from the SR Support Forum (https://www.sr-research.com/support/thread-13.html). For users of Debian-based Linux distributions including Ubuntu, please see the posted installation guide on the SR Support Forum (https://www.sr-research.com/support/docs.php?topic=linuxsoftware). Once you have configured the software repository on your Ubuntu machine, you can run the following commands to install the EyeLink Developers Kit, and of course, the PyLink library.

```
sudo apt install eyelink-display-software
```

3.2 Configure the IP address of the Display PC

A typical setup of an EyeLink tracker involves two computers, a Host PC for eye tracking data registration and a Display PC for stimulus presentation. The Display PC communicates with the Host PC through an Ethernet connection. For successful Ethernet communication between the two computers, the Display PC has to be on the same network as the Host PC. One frequently seen difficulty when connecting to the tracker is caused by failing to set the IP address configuration on the Display PC. Following the EyeLink Installation Guide (https://www.sr-research.com/support/thread-281.html), the Display PC IP address is typically set to 100.1.1.2, and the subnet mask should be 255.255.255.0 – N.B. the network configurations may differ from these defaults in setups with other network-synchronized hardware, most often with EEG integrations. See https://www.sr-research.com/support for more information.

3.3 Install and configure PyLink

3.3.1 Install the PyLink library using *pip*

PyLink is now hosted on the well-known PyPI Python Package Index (*c.f.* https://pypi.org) for the easiest installation on computers with internet connectivity.

```
pip install sr-research-pylink --user
```

In addition to the above online installation, the PyLink library is also included in the EyeLink Developers Kit as Python wheel files (in the *wheels*' folder) for offline installation. Separate wheel files are provided for different versions of Python and architectures. You can use *pip* to install these locally available wheel files by executing a command in a Windows Command prompt or a macOS / Ubuntu terminal (may require administrative privilege). Please replace *.whl with the absolute or relative path to a specific wheel file with the filename tagged accordingly to match the version of python and architecture you plan to use, for example, *sr_research_pylink-2.1.731.0-cp38-cp38-macosx_10_9_universal2.macosx_11_0_universal2.whl*.

```
pip install *.whl –user

# or in expanded form:

pip install sr_research_pylink-2.1.731.0-cp38-cp38-macosx_10_9_universal2.macosx_11_0_universal2.whl --user
```

Please note on macOS the Pylink wheels files that bundled with version 2.1.762 or later of the EyeLink Developers Kit use the "*universal2*" tag so that they support both Intel x86_64 and arm64 architectures. If you are seeing an error message "*universal2.whl is not a supported wheel on this platform", you will need to update pip to the latest version. Please follow Section 1.2 **for instructions using pip installation for the particular Python and operating systems you use.**

3.3.2 Manually install the PyLink library

You can, of course, manually install PyLink by copying the PyLink library to the Python *site-packages* folder (Windows and macOS) or the *dist-package* folder (Ubuntu / Linux). If you are unsure about where to find the *site-package* or *dist-package* folder, open a Python shell, then type in the following commands to show the Python paths. On my MacBook, I have the following output in the Python shell.

```
eyelink@eyelinks-MacBook-Pro ~ % python3.8
Python 3.8.7 (v3.8.7:6503f05dd5, Dec 21 2020, 12:45:15)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>import site
>>>site.getsitepackages()
['/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages']
```

Once you have located the *site-package* folder, navigate to the folder that contains the *pylink* library, then copy the *pylink* library to the *site-packages* folder. On macOS or Linux, use the *cp* command with the *-R* option (copy recursively, i.e., copy the folder and its contents).

```
cp -R pylink /Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages
```

4 Example Scripts

SR Research provides two sets of example scripts as part of the EyeLink Developers Kit. You can find these examples in the following folders.

Windows: C:\Program Files (x86)\SR Research\Eyelink\SampleExperiments\Python\examples
macOS: /Applications/Eyelink/SampleExperiments/Python/examples
Linux: /usr/share/EyeLink/SampleExperiments/Python/examples

4.1 Scripts that do not require a multimedia library

It is possible to run some of the scripts without installing additional Python multimedia libraries (e.g., pygame or experimental design application suites like PsychoPy, OpenSesame etc.). The examples in the *linkEvents*, and *linkSample* folder fall into this category. Please see below for a brief summary of these examples.

- linkEvent -- This script shows the frequently used commands for connecting to the tracker, configuring tracker parameters, starting/ending recording, and messaging for event logging. Most importantly, this script shows how to retrieve eye events (Fixation Start / End, Saccade Start / End, etc.) during data recording from the stimulus presentation PC.
- linkSample -- This script shows the frequently used commands for connecting to the tracker, configuring tracker parameters, starting/ending recording, and messaging for event logging. Most importantly, this script shows how to retrieve samples (time stamped gaze position, pupil size, etc.) in real-time during data recording.

Please note presently the linkSample and linkEvent examples still use calibration graphics that are not compatible with the Arm64 architecture. Therefore, you will see the following error when running the two examples with Python 3.10, 3.11, or 3.12 on Macs that use Apple Silicon M1 or M2 chips. The examples will run fine once the graphics functions are removed.

```
eyelink@eyelinks-MacBook-Pro linkEvent % python3.11 link_event.py

displayAPI: Connecting
Traceback (most recent call last):
  File
"/Users/eyelink/Documents/SampleExperiments_2023_06_19/Python/examples/linkEvent/link_event.
py", line 532, in <module>
    pylink.openGraphics((SCN_WIDTH, SCN_HEIGHT), 32)
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/pylink/eyelink.py", line 1055, in openGraphics
    import pylink.pylink_cg
ImportError: dlopen(/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/pylink/pylink_cg.cpython-311-darwin.so, 0x0002): tried:
'/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/pylink/pylink_cg.cpython-311-darwin.so' (mach-o file, but is an incompatible
architecture (have 'x86_64', need 'arm64')),
'/System/Volumes/Preboot/Cryptexes/OS/Library/Frameworks/Python.framework/Versions/3.11/lib/
python3.11/site-packages/pylink/pylink_cg.cpython-311-darwin.so' (no such file),
'/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/pylink/pylink_cg.cpython-311-darwin.so' (mach-o file, but is an incompatible
architecture (have 'x86_64', need 'arm64'))
```

4.2 Scripts that require PsychoPy or Ppygame

In a typical experimental task, one may use a dedicated library for graphics generation, keyboard response collection, etc. There are lots of free (or open source) Python libraries that one can use for this purpose. Here we provide examples for PsychoPy and pygame to illustrate the eye tracker integration with a Python-based programming tool through the PyLink library.

Note that for both the PsychoPy and pygame examples, there is an accompanying library in the same folder as the example script, e.g., *EyeLinkCoreGraphicsPsychoPy.py*. There are also three .wav files that this .py library depends on. This .py library and the .wav files are needed for tracker calibration (for a brief discussion on this library, see **Section 5.6 Calibration graphics library**). You don’t need to change these files in any way, but please include them in your experimental scripts folder.

Two sets of examples are provided for PsychoPy: Coder and Builder examples (under “Python\examples\Psychopy_examples”). The “Coder” folder provides examples created using only Python code. The “Builder” folder provides some examples for experimental tasks built in PsychoPy’s GUI-based environment by dragging and dropping functional components onto a timeline. In the Builder examples discussed herewith, the EyeLink integration is accomplished using Builder’s “Code Component” with scripted use of PyLink.

In PsychoPy's Builder environment, an alternative option for adding EyeLink integration is to use the EyeLink Plugin for PsychoPy, which provides EyeLink-specific GUI components that perform the EyeLink functions without the use of any Python scripting. Please refer to Getting Started with PsychoPy / EyeLink Integration for more information about the EyeLink Plugin.

A UsingPylinkFromPsychoBuilder.pptx file can also be found in the “Builder” folder – it provides an overview of the Psychopy Builder interface and discusses a few critical components implemented for the proper integration with the EyeLink eye tracker and Data Viewer software.

Please see below for descriptions of the examples in the *PsychoPy_examples* and *pygame_examples* folders.

- fixationWindow_fastSamples / EyeLinkFixationWindowFastSamples_Builder -- This is a basic example, which shows how to implement a gaze-based trigger. We first show a fixation cross at the center of the screen. A trial will move on only when the gaze has been directed to the fixation cross. We then show a picture and wait for the participant to issue a keypress response, or until 5 seconds have elapsed.
- GC_window / EyeLinkGCWindow_Builder -- This example shows how to manipulate the visual stimuli based on real-time gaze data. A mask is shown at the current gaze position in the "mask" condition; in the "window" condition, the image is masked, and a window at the current gaze position will reveal the image hidden behind the mask.
- MRI_demo / EyeLinkMRIdemo_Builder -- This is a basic example illustrating how to do continuous eye tracker recording through a block of trials (e.g., in an MRI setup), and how to synchronize the presentation of trials with a sync signal from the MRI. With a long recording, we start and stop recording at the beginning and end of a testing session (run), rather than at the beginning and end of each experimental trial. We still send the *TRIALID* and *TRIAL_RESULT* messages to the tracker, and Data Viewer will still be able to segment the long recording into small segments (trials).
- picture / EyeLinkPicture_Builder -- This is a basic example, which shows how to connect to and disconnect from the tracker, how to open and close data files, how to start / stop recording, and the standard messages for integration with the Data Viewer software. We show four pictures one-by-one on each trial, and a trial terminates upon a keypress response or until 3 secs have elapsed.
- pursuit / EyeLinkPursuit_Builder -- This example shows how to record the target position in a smooth pursuit task. This script also shows how to record dynamic interest area and target position information to the EDF data file so Data Viewer can recreate the interest area and playback the target movement.
- saccade / EyeLinkSaccade_Builder -- This example shows how to retrieve eye events (saccades) during testing. A visual target appears on the left side, or right side of the screen and the participant is required to quickly shift gaze to look at the target (pro-saccade) or a mirror location on the opposite side of the central fixation (anti-saccade).

The Coder and Builder folders of the *PsychoPy_examples* folder also include an example on video playback.

- video / EyeLinkVideo_Builder -- This example shows how to present video stimuli and how to log frame information in the EDF data file so the gaze data can be correctly laid over the video in Data Viewer

4.3 Run an example script on Windows

4.3.1 Run a PsychoPy example script

If you use the standalone version of PsychoPy, simply open the script in Coder and click the Run button on the toolbar.

4.3.2 Run a pygame example script

You can run a pygame example script from the Command Prompt. Open a Command Prompt, by searching for "cmd" from the Start menu. Assume that we have the *picture* example on the Desktop, the following command will run the *picture.py* script with Python 3.6 from the Command window.

```
py -3.6 Desktop\picture\picture.py
```

You may open the script in the default Python IDE (IDLE) and run the script from there. However, we would not recommend this approach as we have seen performance issues in the past (e.g., slow file transfer at the end of an experiment).

4.4 Run an example script on macOS

On macOS, you can launch a PsychoPy script from the Coder interface of PsychoPy. If you installed PsychoPy as a Python module, you can launch a PsychoPy example script from the terminal.

For pygame, please launch the script from the terminal. For instance, assume I have the *picture* example folder in the *Desktop* folder, the following command will run the picture.py script with Python 3.8. In this command, ~ represents the home folder of the current user.

```
python3.8 ~/Desktop/picture/picture.py
```

4.5 Run an example script on Ubuntu / Linux

On Ubuntu, you can launch an example script from the terminal.

5 Example script walk-throughs

The example scripts are heavily commented. There is no need to go through all the PsychoPy and pygame examples for an introduction to PyLink usage. Here we will walk-through the *picture.py* script from the *PsychoPy_examples/Coder* folder, just to illustrate the basic usage of the PyLink library and the critical steps / protocols for Data Viewer integration.

5.1 Set up an EDF data file name

At the beginning of the script, we first prompt the experimenter to specify an EDF data filename. The filename should not exceed 8 characters and should only contain letters, numbers, and underscores (_). In the script, we have also set up a few folders to store the data file and the resources (e.g., images) we used in the experiment.

```
# Set up EDF data file name and local data folder
#
# The EDF data filename should not exceed 8 alphanumeric characters
# use ONLY number 0-9, letters, & _ (underscore) in the filename
edf_fname = 'TEST'

# Prompt user to specify an EDF data filename
# before we open a fullscreen window
dlg_title = 'Enter EDF File Name'
dlg_prompt = 'Please enter a file name with 8 or fewer characters\n' + \
             '[letters, numbers, and underscore].'

# loop until we get a valid filename
while True:
    dlg = gui.Dlg(dlg_title)
    dlg.addText(dlg_prompt)
    dlg.addField('File Name:', edf_fname)
    # show dialog and wait for OK or Cancel
    ok_data = dlg.show()
    if dlg.OK:  # if ok_data is not None
        print('EDF data filename: {}'.format(ok_data[0]))
    else:
        print('user cancelled')
        core.quit()
        sys.exit()

    # get the string entered by the experimenter
    tmp_str = dlg.data[0]
    # strip trailing characters, ignore the ".edf" extension
    edf_fname = tmp_str.rstrip().split('.')[0]

    # check if the filename is valid (length <= 8 & no special char)
    allowed_char = ascii_letters + digits + '_'
    if not all([c in allowed_char for c in edf_fname]):
        print('ERROR: Invalid EDF filename')
    elif len(edf_fname) > 8:
        print('ERROR: EDF filename should not exceed 8 characters')
    else:
        break
```

5.2 Connect to the tracker

The Display PC connects to the EyeLink Host PC through an Ethernet cable. At the beginning of an eye-tracking experiment, we need to establish an active connection to the Host PC. Without this connection, the experimental script cannot send over commands to control the tracker, nor can it receive gaze data from the eye tracker. The command for initializing a connection is *pylink.EyeLink()*. This command takes just one parameter, the IP address of the Host PC. If you omit the IP address, this method will use the default address of the EyeLink Host PC, which is 100.1.1.1.

The command *pylink.EyeLink()* returns an EyeLink object (i.e., el_tracker in the example code below), which has a set of methods that we can use to control the tracker. Once there is an active connection, the EyeLink Host PC status will switch to “TCP / IP Link Open” (shown in the top-right corner of the Host PC screen).

```
# Step 1: Connect to the EyeLink Host PC
#
# The Host IP address, by default, is "100.1.1.1".
# the "el_tracker" objected created here can be accessed through the PyLink
# Set the Host PC address to "None" (without quotes) to run the script
# in "Dummy Mode"
if dummy_mode:
    el_tracker = pylink.EyeLink(None)
else:
    try:
        el_tracker = pylink.EyeLink("100.1.1.1")
    except RuntimeError as error:
        print('ERROR:', error)
    core.quit()
    sys.exit()
```

The eye-tracker in your lab may not always be available, or you may find it more convenient to debug your experimental script on a computer that is not physically connected to the EyeLink Host PC. If your script does not rely on real-time eye movement data, you can toggle the *dummy_mode* variable to **True** to open a simulated connection to the tracker.

```
# Set this variable to True to run the script in "Dummy Mode"
dummy_mode = False
...

If dummy_mode:
    el_tracker = pylink.EyeLink(None)
```

5.3 Open an EDF data file

The EyeLink Host PC is a machine dedicated to eye-tracking and data logging. At the beginning of a new testing session, we open a data file on the Host PC to store the eye movement data. The data file can be retrieved from the Host PC at the end of a testing session, after closing the data file. For backward compatibility, the EDF file name should not exceed eight characters (not including the .edf extension). To open an EDF data file on the Host PC, use the *openDataFile()* command.

```python
# Step 2: Open an EDF data file on the Host PC
edf_file = edf_fname + ".EDF"
try:
    el_tracker.openDataFile(edf_file)
except RuntimeError as err:
    print('ERROR:', err)
    # close the link if we have one open
    if el_tracker.isConnected():
        el_tracker.close()
    core.quit()
    sys.exit()
```

It is a good practice to put some header information in the EDF data file. Otherwise, it is difficult to tell which EDF data file belongs to which research project.

```python
# Add a header text to the EDF file to identify the current experiment name
# This is OPTIONAL. If your text starts with "RECORDED BY " it will be
# available in DataViewer's Inspector window by clicking
# the EDF session node in the top panel and looking for the "Recorded By:"
# field in the bottom panel of the Inspector.
preamble_text = 'RECORDED BY %s' % os.path.basename(__file__)
el_tracker.sendCommand("add_file_preamble_text '%s'" % preamble_text)
```

5.4 Configure the tracker

One may change the tracker parameters (e.g., sample rate) by clicking the relevant GUI buttons on the EyeLink Host PC. However, a less error-prone approach is to configure tracker parameters by sending commands to the Host PC with the *sendCommand()* method.

```python
# Step 3: Configure the tracker
#
# Put the tracker in offline mode before we change tracking parameters
el_tracker.setOfflineMode()

# Get the software version:  1-EyeLink I, 2-EyeLink II, 3/4-EyeLink 1000,
# 5-EyeLink 1000 Plus, 6-Portable DUO
EyeLink_ver = 0  # set version to 0, in case running in Dummy mode
if not dummy_mode:
    vstr = el_tracker.getTrackerVersionString()
    EyeLink_ver = int(vstr.split()[-1].split('.')[0])
    # print out some version info in the shell
    print('Running experiment on %s, version %d' % (vstr, EyeLink_ver))

# File and Link data control
# what eye events to save in the EDF file, include everything by default
file_event_flags = 'LEFT,RIGHT,FIXATION,SACCADE,BLINK,MESSAGE,BUTTON,INPUT'
# what eye events to make available over the link, include everything by default
link_event_flags = 'LEFT,RIGHT,FIXATION,SACCADE,BLINK,BUTTON,FIXUPDATE,INPUT'
# what sample data to save in the EDF data file and to make available
# over the link, include the 'HTARGET' flag to save head target sticker
# data for supported eye trackers
if EyeLink_ver> 3:
    file_sample_flags = 'LEFT,RIGHT,GAZE,HREF,RAW,AREA,HTARGET,GAZERES,BUTTON,STATUS,INPUT'
    link_sample_flags = 'LEFT,RIGHT,GAZE,GAZERES,AREA,HTARGET,STATUS,INPUT'
else:
    file_sample_flags = 'LEFT,RIGHT,GAZE,HREF,RAW,AREA,GAZERES,BUTTON,STATUS,INPUT'
    link_sample_flags = 'LEFT,RIGHT,GAZE,GAZERES,AREA,STATUS,INPUT'
el_tracker.sendCommand("file_event_filter = %s" % file_event_flags)
el_tracker.sendCommand("file_sample_data = %s" % file_sample_flags)
el_tracker.sendCommand("link_event_filter = %s" % link_event_flags)
el_tracker.sendCommand("link_sample_data = %s" % link_sample_flags)

# Optional tracking parameters
# Sample rate, 250, 500, 1000, or 2000, check your tracker specification
# if EyeLink_ver> 2:
#     el_tracker.sendCommand("sample_rate 1000")
# Choose a calibration type, H3, HV3, HV5, HV13 (HV = horizontal/vertical),
el_tracker.sendCommand("calibration_type = HV9")
# Set a gamepad button to accept calibration/drift check target
# You need a supported gamepad/button box that is connected to the Host PC
el_tracker.sendCommand("button_function 5 'accept_target_fixation'")
```

5.5 Open a window

We need to open a window to present the visual stimuli and to calibrate the tracker. Note here, we need to send the correct screen resolution to the Host PC with the *screen_pixel_coords* command and log this info in the EDF data file with a *DISPLAY_COORDS* message. The *DISPLAY_COORDS* message will ensure the Trial View window is sized appropriately in Data Viewer.

```
# Open a window, be sure to specify monitor parameters
mon = monitors.Monitor('myMonitor', width=53.0, distance=70.0)
win = visual.Window(fullscr=full_screen,
                    monitor=mon,
                    winType='pyglet',
                    units='pix')
```

```
# get the native screen resolution used by PsychoPy
scn_width, scn_height = win.size
# resolution fix for Mac retina displays
if 'Darwin' in platform.system():
    if use_retina:
        scn_width = int(scn_width/ 2.0)
        scn_height = int(scn_height / 2.0)

# Pass the display pixel coordinates (left, top, right, bottom) to the tracker
# see the EyeLink Installation Guide, "Customizing Screen Settings"
el_coords = "screen_pixel_coords = 0 0 %d %d" % (scn_width - 1, scn_height - 1)
el_tracker.sendCommand(el_coords)

# Write a DISPLAY_COORDS message to the EDF file
# Data Viewer needs this piece of info for proper visualization, see Data
# Viewer User Manual, "Protocol for EyeLink Data to Viewer Integration"
dv_coords = "DISPLAY_COORDS 0 0 %d %d" % (scn_width - 1, scn_height - 1)
el_tracker.sendMessage(dv_coords)
```

One thing worth noting here is that there are a few lines of code that help to fix drawing issues in PsychoPy when running on macOS with retina displays. If you are using PsychoPy and a retina display, or you are connecting a Macbook laptop to an external monitor, please see **Section 6.1.1** for known issues and troubleshoot tips.

5.6 Calibration graphics library

With the EyeLink tracker, calibration is a two-step process: calibrating the tracker and evaluating the calibration results using a validation procedure. Both steps involve presenting a series of visual targets at different known screen positions. The observer is required to shift gaze to follow the calibration target. The calibration process can use 3, 5, 9, or 13 screen positions. A 9-point calibration (HV-9) will give you the best results if you use a chin rest to stabilize the head of the observer. A 5-point (HV5) or 13-point calibration (HV13) will provide you with better results when the head is free to move (i.e., tracking in Remote Mode). Following calibration, there is a validation process in which a visual target appears at known screen positions, and the observer shifts gaze to follow the target. By comparing the gaze position reported by the tracker and the physical position of the target, the tracker can estimate the gaze errors at different screen positions. This two-step procedure (calibration and validation) requires a single PyLink command, e.g.,

```
el_tracker.doTrackerSetup()
```

We need to configure the calibration graphics (window) before we call this command. Note that for both the PsychoPy and pygame examples, there is an accompanying library in the same folder as the example script (*EyeLinkCoreGraphicsPsychoPy.py* or *CalibrationGraphicsPygame.py*). There are also three .wav files that this .py library depends on.

Please do not change these files unless you know exactly what you are doing. The *EyeLinkCoreGraphicsPsychoPy.py* is frequently referred to as a CoreGraphics library; it implements a set of methods that will be used during calibration, validation, and drift-correction etc. So, when the calibration routine is evoked, PyLink will use the methods defined in this library to draw the camera image, the calibration/validation target, play the warning beeps, etc. To use this library, we need to first import the *EyeLinkCoreGraphicsPsychoPy* class from it.

```
From EyeLinkCoreGraphicsPsychoPy import EyeLinkCoreGraphicsPsychoPy
```

Then, in the experimental script we create an instance of the *EyeLinkCoreGraphicsPsychoPy* class, e.g.,

```
genv = EyeLinkCoreGraphicsPsychoPy(el_tracker, win)
```

Here, we pass the tracker connection (el_tracker) and the window we plan to use for calibration when initializing the new *EyeLinkCoreGraphicsPsychoPy* instance. Then, we configure its various parameters, e.g., foreground/background color, calibration target. Note that the calibration target could be a "circle" (default), a "picture", a "movie" clip, or a dynamically rotating "spiral". To configure the type of calibration target, simple set *genv.setTargetType()* to "circle", "picture", "movie", or "spiral", e.g., *genv.setTargetType('picture')*. You may configure the warning beeps as well. You can use the default warning beeps (shown in the example script below) or use a custom .wav file as the warning beep.

```
# Configure a graphics environment (genv) for tracker calibration
genv = EyeLinkCoreGraphicsPsychoPy(el_tracker, win)
print(genv)  # print out the version number of the CoreGraphics library

# Set background and foreground colors for the calibration target
# in PsychoPy, (-1, -1, -1)=black, (1, 1, 1)=white, (0, 0, 0)=mid-gray
foreground_color = (-1, -1, -1)
background_color = win.color
genv.setCalibrationColors(foreground_color, background_color)

# Set up the calibration target
#
# The target could be a "circle" (default), a "picture", a "movie" clip,
# or a rotating "spiral". To configure the type of calibration target, set
# genv.setTargetType to "circle", "picture", "movie", or "spiral", e.g.,
# genv.setTargetType('picture')
#
# Use gen.setPictureTarget() to set a "picture" target
# genv.setPictureTarget(os.path.join('images', 'fixTarget.bmp'))
#
# Use genv.setMovieTarget() to set a "movie" target
# genv.setMovieTarget(os.path.join('videos', 'calibVid.mov'))
# Use a picture as the calibration target
genv.setTargetType('picture')
genv.setPictureTarget(os.path.join('images', 'fixTarget.bmp'))

# Configure the size of the calibration target (in pixels)
# this option applies only to "circle" and "spiral" targets
# genv.setTargetSize(24)

# Beeps to play during calibration, validation and drift correction
# parameters: target, good, error
#     target -- sound to play when target moves
#     good -- sound to play on successful operation
#     error -- sound to play on failure or interruption
# Each parameter could be ''--default sound, 'off'--no sound, or a wav file
genv.setCalibrationSounds('', '', '')

# resolution fix for macOS retina display issues
if use_retina:
    genv.fixMacRetinaDisplay()

# Request Pylink to use the PsychoPy window we opened above for calibration
pylink.openGraphicsEx(genv)
```

To request PyLink to use the CoreGraphics library for calibration, we need to also call the pylink.openGraphicsEx() command.

```
# Request Pylink to use the PsychoPy window we opened above for calibration
pylink.openGraphicsEx(genv)
```

5.7 Helper functions

The script then defines a few helpful functions to clear the screen, to terminate the task, to register keypresses, show messages, to abort a trial, and to run a single trial. We will go through the functions for terminating the task and running through a trial in a later section.

```
# define a few helper functions for trial handling

def clear_screen(win):
""" clear up the PsychoPy window"""


def show_msg(win, text, wait_for_keypress=True, any_key_to_terminate=True):
""" Show task instructions on screen"""


def terminate_task():
""" Terminate the task gracefully and retrieve the EDF data file

file_to_retrieve: The EDF on the Host that we would like to download
```

```
    win: the current window used by the experimental script
    """


def abort_trial():
"""Ends recording """


def run_trial(trial_pars, trial_index):
""" Helper function specifying the events that will occur in a single trial

trial_pars - a list containing trial parameters, e.g.,
                ['cond_1', 'img_1.jpg']
trial_index - record the order of trial presentation in the task
    """
```

5.8 Calibrate the tracker

To calibrate the tracker, we need to call the *doTrackerSetup*() command. Calibration is usually performed at the beginning of a testing session, at the beginning of a new block of trials, or between trials if tracking accuracy is not ideal.

```
# Step 5: Set up the camera and calibrate the tracker

# Show the task instructions
task_msg = 'In the task, you may press the SPACEBAR to end a trial\n' + \
'\nPress Ctrl-C to if you need to quit the task early\n'
if dummy_mode:
    task_msg = task_msg + '\nNow, press ENTER to start the task'
else:
    task_msg = task_msg + '\nNow, press ENTER to calibrate tracker'
    show_msg(win, task_msg, wait_for_keypress=False)

# skip this step if running the script in Dummy Mode
if not dummy_mode:
    try:
        el_tracker.doTrackerSetup()
    except RuntimeError as err:
        print('ERROR:', err)
        el_tracker.exitCalibration()
    Should_recal = 'no'
```

5.9 Run through all trials

The parameters of each trial are put into a list, at the beginning of the script. Following calibration, we loop over the list that contains the parameters of all trials to run through all the trials.

```
# Store the parameters of all trials in a list, [condition, image]
trials = [
    ['cond_1', 'img_1.jpg'],
    ['cond_2', 'img_2.jpg'],
    ]

# Step 6: Run the experimental trials, index all the trials

# construct a list of 4 trials
test_list = trials[:]*2

# randomize the trial list
random.shuffle(test_list)

trial_index = 1
for trial_pars in test_list:
    run_trial(trial_pars, trial_index)
    trial_index += 1
```

5.10 The *run_trial*() function

In the script, we defined a *run_trial*() function to group the commands we need to execute on each trial. The very first command we executed here will return the link connection to the tracker.

```
el_tracker = pylink.getEYELINK()
```

5.10.1 Backdrop on the Host

We send over commands to clear up the Host PC screen, and then draw the backdrop or landmarks there. This is optional, but can be a helpful feature if you would like to monitor gaze during testing. For drawing a backdrop on the Host PC screen, here we illustrate two different commands, *imageBackdrop()* and *bitmapBackdrop()*. The latter one should work with all versions of the EyeLink Host PC; the former command will work on more recent models (EyeLink 1000 Plus, Portable Duo) and with versions of the EyeLink Developers Kit 2.0 and up. Since the *imageBackdrop()* calls a few methods in the PIL module, you'll need to have the PIL module installed beforehand (by running *pip install pillow*).

A few additional primitive drawing commands (e.g., lines, boxes) supported by the Host PC are documented in the COMMANDS.INI file on the Host PC. These commands provide a lightweight solution for drawing reference landmarks on the Host PC screen. For instance, the correct saccade target location in a pro- / anti-saccade task.

```
# put the tracker in the offline mode first
```

```
el_tracker.setOfflineMode()

# clear the host screen before we draw the backdrop
el_tracker.sendCommand('clear_screen 0')

# show a backdrop image on the Host screen, imageBackdrop() the recommended
# function, if you do not need to scale the image on the Host
# parameters: image_file, crop_x, crop_y, crop_width, crop_height,
#             x, y on the Host, drawing options
##    el_tracker.imageBackdrop(os.path.join('images', pic),
##                             0, 0, scn_width, scn_height, 0, 0,
##                             pylink.BX_MAXCONTRAST)

# If you need to scale the backdrop image on the Host, use the old PyLink
# bitmapBackdrop(), which requires an additional step of converting the
# image pixels into a recognizable format by the Host PC.
# pixels = [line1, ...lineH], line = [pix1,...pixW], pix=(R,G,B)
#
# the bitmapBackdrop() command takes time to return, not recommended
# for tasks where the ITI matters, e.g., in an event-related fMRI task
# parameters: width, height, pixel, crop_x, crop_y,
#             crop_width, crop_height, x, y on the Host, drawing options
#
# Use the code commented below to convert the image and send the backdrop
im = Image.open('images' + os.sep + pic)  # read image with PIL
im = im.resize((scn_width, scn_height))
img_pixels = im.load()  # access the pixel data of the image
    pixels = [[img_pixels[i, j] for i in range(scn_width)]
for j in range(scn_height)]
el_tracker.bitmapBackdrop(scn_width, scn_height, pixels,
                              0, 0, scn_width, scn_height,
                              0, 0, pylink.BX_MAXCONTRAST)

# OPTIONAL: draw landmarks and texts on the Host screen
# In addition to backdrop image, You may draw simples on the Host PC to use
# as landmarks. For illustration purpose, here we draw some texts and a box
# For a list of supported draw commands, see the "COMMANDS.INI" file on the
# Host PC (under /elcl/exe)
    left = int(scn_width/2.0) - 60
    top = int(scn_height/2.0) - 60
    right = int(scn_width/2.0) + 60
    bottom = int(scn_height/2.0) + 60
draw_cmd = 'draw_filled_box %d %d %d %d 1' % (left, top, right, bottom)
el_tracker.sendCommand(draw_cmd)
```

5.10.2 *TRIALID* message

The next step is to send a *TRIALID* message to mark the onset of a new trial. This message is needed for the Data Viewer software to correctly segment the data file into “trials”.

```
# send a "TRIALID" message to mark the start of a trial, see Data
# Viewer User Manual, "Protocol for EyeLink Data to Viewer Integration"
```

```
el_tracker.sendMessage('TRIALID %d' % trial_index)
```

5.10.3 Record status message

It can be helpful to have some information about the current trial on the Host PC, so the experimenter can easily monitor the progress of the task and ensure the participant is actively engaging in the task. In the example script, the status message shows the current trial number, in the bottom-right corner of the Host PC screen.

```
# record_status_message : show some info on the Host PC
# here we show how many trial has been tested
status_msg = 'TRIAL number %d' % trial_index
el_tracker.sendCommand("record_status_message '%s'" % status_msg)
```

5.10.4 Drift-check / drift-correction

One of the most critical commands illustrated in this example script is *doDriftCorrect()*. Following this command, a target appears on the screen, and the participant needs to look at the target and then (the experimenter or the participant) confirm the gaze position by pressing a key. The tracker will estimate the current tracking accuracy with the reported gaze position and the physical position of the target.

This feature is known as drift-correction or drift-check, and one can think of it as a 1-point validation of tracking accuracy. The concept of drift correction is inherited from earlier head-mounted versions of the EyeLink eye-tracker (EyeLink I and II). In these models, the headband could slip and cause drifts in the gaze data, especially when operating the eye tracker in the pupil-only tracking mode. This issue was tackled by a linear correction of the gaze data based on the gaze error reported by the drift-correction procedure. For recent EyeLink eye trackers (EyeLink 1000, 1000 Plus, and Portable Duo), by default, the gaze error is no longer used to correct the gaze data, as the systems are resilient to small head or camera displacement when performing recording in the pupil-CR mode. Instead, the drift-correction routine only performs a “drift-check”; it checks the tracking accuracy and allows users to recalibrate if necessary.

```
# drift check
# we recommend drift-check at the beginning of each trial
# the doDriftCorrect() function requires target position in integers
# the last two arguments:
# draw_target (1-default, 0-draw the target then call doDriftCorrect)
# allow_setup (1-press ESCAPE to recalibrate, 0-not allowed)
#
# Skip drift-check if running the script in Dummy Mode
while not dummy_mode:
    # terminate the task if no longer connected to the tracker or
    # user pressed Ctrl-C to terminate the task
    if (not el_tracker.isConnected()) or el_tracker.breakPressed():
        terminate_task()
        return pylink.ABORT_EXPT

    # drift-check and re-do camera setup if ESCAPE is pressed
    try:
        error = el_tracker.doDriftCorrect(int(scn_width/2.0),
                                          int(scn_height/2.0), 1, 1)
        # break following a success drift-check
        if error is not pylink.ESC_KEY:
            break
    except:
        pass
```

The *doDriftCorrect*() command takes four parameters. The first two are x, y pixel coordinates for the drift-check target. Note that x, y must be integers, (e.g., 512, 384). The third parameter specifies whether PyLink should draw the target. If set to 0, we need to first draw a custom target at the x, y pixel coordinates, then call the *doDriftCorrect*() command. The fourth parameter controls whether PyLink should evoke the calibration routine if the ESCAPE key is pressed.

5.10.5 Data recording

The most common recording implementation is to start recording at the beginning of each trial and stop recording at the end of each trial. By doing so, the tracker does not record data during the inter-trial intervals, reducing the size of the EDF data file, and allowing performing a drift check and/or update the Host PC backdrop graphics. For situations where a single continuous recording is preferred (e.g., in an fMRI or EEG study), one can start data recording at the beginning of a session (or run) and stop data recording at the end of a session, and use messages (e.g, *TRIALID* and *TRIAL_RESULT*) to mark the beginning and endpoints of trials within the continuous recording.

To start data recording, call *startRecording()*. This command takes four parameters specifying what types of data (event or sample) are recorded in the EDF data file and what types of data are available over the link during testing. With the parameter (1, 1, 1, 1), the tracker will record both events and samples in the data file and also make these two types of data available over the link.

```
# put tracker in idle/offline mode before recording
el_tracker.setOfflineMode()

# Start recording
# arguments: sample_to_file, events_to_file, sample_over_link,
# event_over_link (1-yes, 0-no)
try:
    el_tracker.startRecording(1, 1, 1, 1)
```

```
except RuntimeError as error:
    print("ERROR:", error)
    abort_trial()
    return pylink.TRIAL_ERROR
```

5.10.6 Logging messages

Messages are sent to the tracker whenever a critical event occurs; for instance, a picture appears on the screen. With these messages in the EDF data file, we can tell what events occurred during testing, and segment the recording for meaningful analysis (e.g., by setting Interest Periods in the Data Viewer software). The example script sends the *image_onset* message to the tracker immediately after a picture appears on the screen.

```
el_tracker.sendMessage('image_onset')
```

In addition to messages that mark critical trial events, we also log messages that will facilitate data visualization and analysis in the Data Viewer software. The expected formatting and the options available for the functions performed by these special messages are described in the Protocol for EyeLink Data to Viewer Integration (https://www.sr-research.com/support/thread-83.html). The first such message illustrated in the script is *IMGLOAD*. This message points to the picture presented during testing; when Data Viewer sees this message, it will find the picture and draw as the background in the Trial View window's Spatial Overlay and Animation Views.

```
# Send a message to clear the Data Viewer screen, get it ready for
# drawing the pictures during visualization
bgcolor_RGB = (116, 116, 116)
el_tracker.sendMessage('!V CLEAR %d %d %d' % bgcolor_RGB)

# send over a message to specify where the image is stored relative
# to the EDF data file, see Data Viewer User Manual, "Protocol for
# EyeLink Data to Viewer Integration"
bg_image = '../../images/' + pic
imgload_msg = '!V IMGLOAD CENTER %s %d %d %d %d' % (bg_image,
                                                    int(scn_width/2.0),
                                                    int(scn_height/2.0),
                                                    int(scn_width),
                                                    int(scn_height))
el_tracker.sendMessage(imgload_msg)
```

The second message illustrated here is an Interest Area defining message, which will be used to create Interest Areas in Data Viewer.

```
# send interest area messages to record in the EDF data file
# here we draw a rectangular IA, for illustration purposes
# format: !V IAREA RECTANGLE <id><left><top><right><bottom> [label]
# for all supported interest area commands, see the Data Viewer Manual,
# "Protocol for EyeLink Data to Viewer Integration"
ia_pars = (1, left, top, right, bottom, 'screen_center')
el_tracker.sendMessage('!V IAREA RECTANGLE %d %d %d %d %d %s' % ia_pars)
```

A third set of messages are illustrated here to show how to record variables and values into the EDF data file, so Data Viewer can recognize the variables automatically when loading the EDF data file.

```
# record trial variables to the EDF data file, for details, see Data
# Viewer User Manual, "Protocol for EyeLink Data to Viewer Integration"
el_tracker.sendMessage('!V TRIAL_VAR condition %s' % cond)
el_tracker.sendMessage('!V TRIAL_VAR image %s' % pic)
el_tracker.sendMessage('!V TRIAL_VAR RT %d' % RT)
```

5.10.7 The *TRIAL_RESULT* message

At the end of a trial, we include the following line of code to send a *TRIAL_RESULT* message. Here, we add a flag "0" to the *TRIAL_RESULT* message to indicate the trial completed successfully. This message marks the end of a trial. Before we send over this message, it is recommended to send the *!V CLEAR* message. This message will clear any drawing in Data Viewer by the end of a trial. The *TRIAL_RESULT* message should be sent, following the *stopRecording()* command, so the recording is enclosed in a pair of *TRIALID* and *TRIAL_RESULT* messages, which Data Viewer uses to segment the trials.

```
# send a message to clear the Data Viewer screen as well
el_tracker.sendMessage('!V CLEAR 128 128 128')

# stop recording; add 100 msec to catch final events before stopping
pylink.pumpDelay(100)
el_tracker.stopRecording()

# send a 'TRIAL_RESULT' message to mark the end of trial, see Data
# Viewer User Manual, "Protocol for EyeLink Data to Viewer Integration"
el_tracker.sendMessage('TRIAL_RESULT %d' % pylink.TRIAL_OK)
```

5.10.8 The *terminate_task()* function

In the script, we defined a *terminate_task()* function to perform the housekeeping job at the end of a testing session, or in case that the experimenter terminated the task prematurely. Three jobs are done in this function, namely put the tracker in Offline mode, close the data file, and download the file to the stimulus presentation PC and terminate the link to the tracker. It is recommended to put the tracker in Offline mode and add a delay before closing the data file. This helps to protect data integrity.

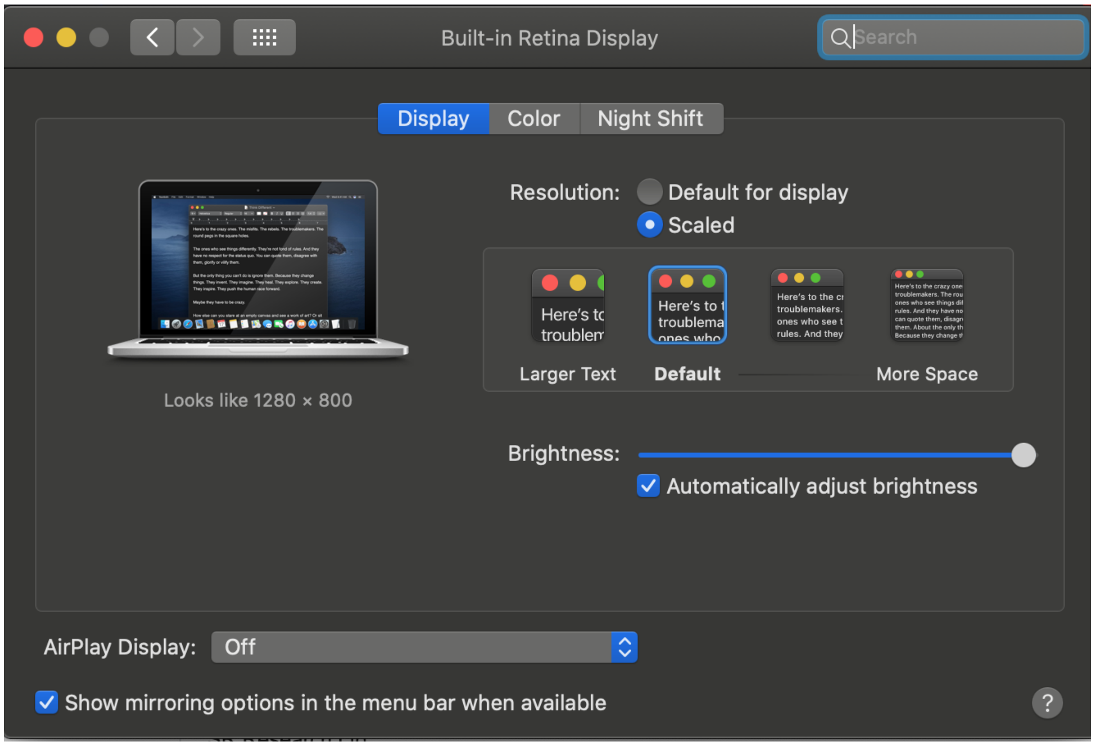
```python
if el_tracker.isConnected():
    # Terminate the current trial first if the task terminated prematurely
    error = el_tracker.isRecording()
    if error == pylink.TRIAL_OK:
        abort_trial()

    # Put tracker in Offline mode
    el_tracker.setOfflineMode()

    # Clear the Host PC screen and wait for 500 ms
    el_tracker.sendCommand('clear_screen 0')
    pylink.msecDelay(500)

    # Close the edf data file on the Host
    el_tracker.closeDataFile()

    # Show a file transfer message on the screen
    msg = 'EDF data is transferring from EyeLink Host PC...'
    show_msg(win, msg, wait_for_keypress=False)

    # Download the EDF data file from the Host PC to a local data folder
    # parameters: source_file_on_the_host, destination_file_on_local_drive
    local_edf = os.path.join(session_folder, session_identifier + '.EDF')
    try:
        el_tracker.receiveDataFile(edf_file, local_edf)
    except RuntimeError as error:
        print('ERROR:', error)

    # Close the link to the tracker.
    el_tracker.close()
```

6 Known issues and trouble-shooting tips

6.1 PsychoPy issues

6.1.1 Retina displays

The high-res retina displays on macOS may give people issues when using PsychoPy. One easy solution is to use the “scaled down” resolution instead of the native pixel resolution of the retina display. If you go to the Retina Display settings, tick the “Scaled” option, and hover the mouse on the option of your choice (e.g., Default). You will see the scaled-down resolution on the left side (e.g., 1280 x 800 in the screenshot below). This is the resolution PsychoPy will use to draw your stimuli. However, if you read the “size” of the full screen window you opened PsychoPy will report the actual pixel resolution (e.g., 2560 x 1600). Users should be aware of this discrepancy.


Built-in Retina Display

Display | Color | Night Shift

Resolution: Default for display [✓ Scaled]

Larger Text | **Default** | More Space

Looks like 1280 × 800

Brightness:

[✓] Automatically adjust brightness

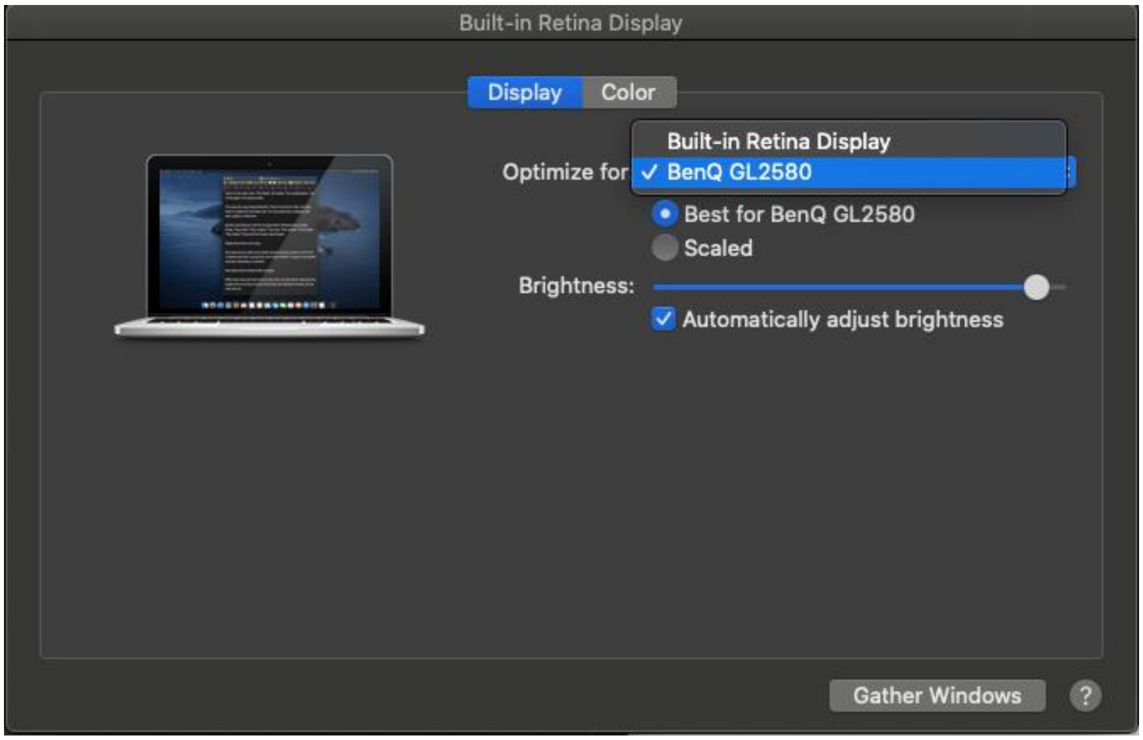
AirPlay Display: Off

[✓] Show mirroring options in the menu bar when available

If you are using an external monitor connected to a MacBook (equipped with retina display), macOS will allow you to either optimize screen resolution for the built-in retina display or the external monitor.

In the PsychoPy Coder example scripts, we added a *use_retina* variable to help avoid the various issues related to retina screen resolution. If you choose to optimize the screen resolution for the external monitor, please be sure to set the *use_retina* variable in the example script to **False**. If you choose to optimize the screen resolution for the built-in retina display, or the MacBook is not connected to an external monitor, set the *use_retina* variable to **True**.

```
# Set this variable to True if you use the builtin retina screen as your
# primary display device on macOS. If you have an external monitor, set this
# variable True if you choose to "Optimize for Built-in Retina Display"
# in the Displays preference settings.
use_retina = False
```



Optimize for: Built-in Retina Display / [✓ BenQ GL2580]

[● Best for BenQ GL2580] Scaled

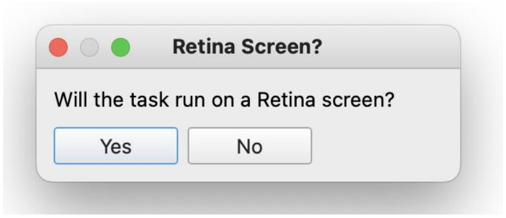
Brightness:

[✓] Automatically adjust brightness

In the PsychoPy example scripts, we also the following command to request the *EyeLinkCoreGraphicsPsychoPy.py* library to use the scaled down resolution if *use_retina* is set to **True**.

```
# resolution fix for macOS retina display issues
if use_retina:
    genv.fixMacRetinaDisplay()
```

The Psychopy Builder examples handle the "use_retina” setting automatically through the following dialog box at the beginning of the experiment run time.



Will the task run on a Retina screen?

6.1.2 PsychoPy window loses focus

We have seen an issue on both Windows and macOS where, in the video example, the screen would lose focus and stop accepting keyboard input when loading the video files. The mouse cursor would show up as well in this case. You need to click the mouse once to get the PsychoPy window in focus. This is a known PsychoPy issue yet to be fixed at time of this documentation (https://discourse.psychopy.org/t/loading-a-movie-somehow-brings-up-the-mouse-cursor-in-fullscreen-mode/19826).

6.1.3 PsychoPy keyboard locks up on macOS

On macOS, we have seen random keyboard lockup when running the example scripts. The script appears to freeze and will not accept keyboard inputs. We suspect this issue arises from some conflicts between the macOS Privacy settings (e.g., “Input Monitoring”) and PsychoPy. We do not have a solution or workaround for this issue yet, but the issue would usually disappear if you re-run the script.

6.2 pygame issues

6.2.1 pygame not accepting any keyboard inputs

We have seen that sometimes the pygame window would lose focus and will not accept any keyboard input. An easy fix to this issue is to click the mouse once. This issue is only seen on Windows with pygame 2.x We have not seen this issue on macOS (with pygame 2.x) or in older versions of pygame (1.9.6).

6.2.2 The “pygame parachute” error

On macOS, you may see the following error messages, especially when using pygame 1.9.6. When this happens, pygame would crash and the scripts would not run. There is unfortunately no proper fix to this issue. You may try a different version of pygame (e.g., a beta build), but it is difficult to tell which version works on which macOS. We have also seen this issue in our tests of pygame 2.x, but it is not as common.

```
Fatal Python error: (pygame parachute) Segmentation Fault Aborted
```