Designing experiments with Python

Using Pytrack to run your eyetracking experiment

Johannes Steger

jss@coders.de

Cliona Quigley

cquigley@uos.de

Designing experiments with Python: Using Pytrack to run your eyetracking experiment

by Johannes Steger and Cliona Quigley

Table of Contents

List of Figures

Chapter 1. Introduction

Pytrack is a Python library written by Johannes for use with SR-Research's Eyelink system, and builds on Pylink, a library provided by SR-Research. It allows users to easily program experiments and to explicitly control the trial and experiment information (meta-information) that will be saved during the experiment.

This manual will equip you with all you need to set up an eyetracking experiment using pytrack, from programming basics to things you should keep in mind when designing an experiment. It is divided into three parts:

Python Crashcourse

Programming basics: enough information to let you get started, and tips on where to find out more.

Eyetracking Background

An overview of the components of our working eyetracking system and how they communicate.

Designing an Experiment

A simple but thorough example of how to use pytrack to run an experiment.

Pylink is not free software, but thanks to Pytrack, you hopefully won't need to worry about what it does.

Chapter 2. Python Crashcourse

2.1. Python - what's the big deal?

Python is not only a powerful programming language, it's also relatively straightforward to use. You can read all about the pros and cons and ins and outs of Python in many books and websites - for now all you need to know is that Python is a good solid tool for programming your eyetracking experiment, and that it will make the process as easy and pain-free as possible!

In the following pages, we'll go through some of the basics by example, but if you want to take the time to do a proper tutorial, you'll find good resources at python.org (http://python.org/) (the Beginner's Guide there is recommended for those new to programming and the Library Reference is of course useful for everyone). If you're already experienced in programming with another language, try diveintopython.org (http://diveintopython.org/).

All you need to write Python programs is an editor (vi, emacs, textpad, etc.), and if you want to run your programs on your machine, you also need an interpreter. All computers in the lab should have an interpreter already installed. IPython is also a useful tool - it is an enhanced Python shell (the 'I' stands for interactive) - google it if you want to know more. You'll find the standard shell on most lab computers - just type 'python' in a terminal window. And now we'll begin...

2.2. The Crash Course

The first important feature of Python that we'll cover is the issue of data types. Unlike most other programming languages, Python does not require the user to state the type of a variable the first time it is used. If you are familiar with Java, you'll know that if you want, for example, to have a variable with the value 4, you first need to declare and initialise it:

```
// THE JAVA WAY OF DOING THINGS:
// variable foo has type integer and is set to 4
int foo = 10;
String blah = "lala";      // blah is a string with value "lala"
```

With Python, life is much easier. Simply assign the values you want to the desired variables. This can be done in a program, written in any editor and saved as `your_program_name.py`, or in a Python shell. For this example, I started a Python shell and entered the following lines:

```
>>> foo = 10
>>> blah = "lala"
```

There is no need to declare the type of each variable; types are implied in your use of the variables and Python deals with them accordingly. You can of course query the type of a variable using `type()`, for example:

```
>>> type(foo)
<type 'int'>
```

This is known as "duck typing" - if it walks like a duck and if it quacks like a duck, then as far as Python is concerned, it is a duck. In other words, if the interpreter expects e.g. an integer, it will just treat your variable as an integer. This is nice, because you can do most things you can do with integers with floats, complex numbers and so on, too - they're walking and quacking the same way. However, if you do anything illogical or illegal with those ducks, expect errors.

Example 2-1. Shooting the duck

```
>>> print foo/blah
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Here, an error arises because it's not acceptable to divide an integer by a string.

You might have noticed that there is no special character used to terminate each line of our program so far. Python does not require a semi-colon (;) at the end of every line of code - in fact, most of the syntax of a program is provided by indenting and normal line-breaks (pressing enter). The colon (:) is another important character. The next example shows you how to write a for loop, this time in a program we'll call `example1.py`.

```
a = 10
b = [1, 2, 0]

for divi in b:
    try:
        print a/divi
    except:
        print "you lose"
```

This program initializes an integer a and list b. The for statement iterates over the items in a sequence given by the programmer - here, we iterate over the members of the list b. For each member of the list, we then attempt to print (to the command line of the shell) the result of the calculation "a divided by current list member". You may be familiar with exceptions or try and catch statements from other langauges; in Python, the try clause is executed first. If any exceptions (errors) occur, then the except clause is executed and no error appears at the command line.

Executing this code (type `python example1.py` in a terminal window) should give the following result:

```
10
5
you lose
```

(Remember that anything divided by 0 is undefined!)

> **Note:** It is very important that you indent correctly, by pressing space 4 times. Messy indenting will most certainly lead to errors... Remember - 4 x space!

To show you how intuitive the rest of the language is, let's look at the example of an if statement. Again, it's crucial that you indent correctly! Let's call this program `example2.py`

```
a = 1
b = "hallo"
c = [1, 2, 3]

if a>0:
    print "a is biiiiig"
else:
    print "a is smaaaaall"
print "done"
```

Executing this program (by typing `python example2.py`, of course) should give:

```
a is biiiiig
done
```

The first condition of the if statement (a>0) was satisfied, so the requested output string was printed. Finally, the string "done" was printed - if the final line of code in our program had been indented, this would only have happened if a was smaller than 0.

Let's extend this program a little to illustrate some other important features. We might want to define a function in our program. A trivial example is a function that prints its input. We already know a statement that does this - `print`. So, let's define our function in terms of what we already have.

```
def func(param):
    print param

a = 1
b = "hallo"
c = [1, 2, 3]

if a>0:
    func("a is biiiiig")
else:
    func("a is smaaaaall")
func("done")
```

That's all there is to it. We've defined a function called `func`, which takes some input `param`, and simply passes this input to the function `print`. It's defined at the top of our program, so now let's add some more lines of code to the end of our program to test `func`.

```
for d in c:
    func(b[d])
```

Now executing the program gives the following output:

```
a is biiiiig
done
a
l
l
```

As before, a>0 is satisfied, and the `print "done"` command is executed. Our new code iterated through the members of list c, i.e. the integers 1, 2 and 3, and passed the corresponding index of the string b to our function func. As is standard in all programming languages (except Matlab!), lists and arrays are indexed beginning with index 0, so the characters of "hallo" at indices 1, 2, and 3 are a, l, and l, respectively.

The last topic we'll deal with in this section is objects. Python is a generic programming language that is perfectly suited to object-oriented programming. We could define the attributes of a particular fruit, an apple, in the following way (which is deliberately not very object-oriented!):

```
apple_diameter = 5.3
apple_color = "light green"
apple_radius = apple_diameter / 2
```

If we then have a second apple we'd like to describe, things get a bit tricky.

So let's do it properly, and define a class of fruits, aptly called `Fruit`, in a program called `AppleClass.py`.

```
class Fruit:
    pass
```

Nothing really happens in this class - the `pass` statement basically does nothing. Next we'll extend this class to make a new class called `Apple`. This class will inherit from its superclass of `Fruit`, will have a default color and diameter, and a method which will allow us to access one of its attributes.

```
class Apple(Fruit):
    color = "grey"
    diameter = 5.3
    def get_radius(self):
        """
        returns the half diameter of this apple
        """
        return self.diameter / 2
```

As you can probably guess from the name, `self` is the equivalent of `this` in Java, and refers to the object itself. To use this class in a program, you must first import the class:

```
import AppleClass
```

Now you can create new instances of this imported class and do what you want with them!

```
ap1 = AppleClass.Apple()
```

```
ap1.color = "red"
```

Finally, a note on documenting your Python code. If you want to include comments in your program, you can use the hash symbol (#), for example:

```
mycounter = 3     # initialise this to 3 for some good reason
```

In the Apple class definition above, we wrote a single class method, `get_radius`. A description of what this method does is given between two lines of three quotation marks ("). This syntax assigns your description to the documentation of that class method. Now if a user (who has correctly imported your apple class and created an instance ap1) types `help(ap1.get_radius)`, the following output will appear:

```
Help on method get_radius in module AppleClass:

get_radius(self) method of AppleClass.Apple instance
    returns the half diameter of this apple
```

Chapter 3. Eyetracking Background

3.1. System Components & Information Flow

As you've probably already noticed, there are two computers in the eyetracking room: the tracker computer (also known as the host), and the display computer. During an experiment, the display computer does exactly what its name says, it displays the stimuli to the subject. The tracker computer deals with all of the processing needed by the eyetracking hardware, i.e. the cameras. Along with the subject who is taking part in your experiment, and possibly a keypad or other input device, the host and display computers and the eyetracker itself constitute the complete eyetracking system.

Figure 3-1. Systemcomponents and Dataflow

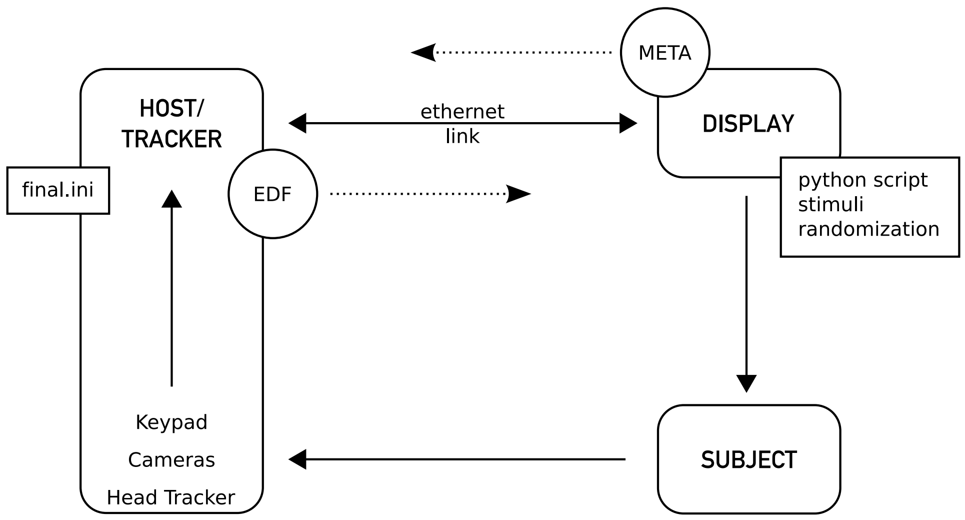

Figure 1 provides a schematic overview of how these components communicate with one another. The two computers are connected with an ethernet link. An experiment (i.e. your Python program) is always started from the display computer. Before a trial begins, the stimuli are prepared for presentation and messages relating to the experiment (meta-data) may be sent to the tracker computer to be written to the edf file. Depending on the experiment, a subject might give some feedback by means of the keypad which is also written to file. During a trial, the data from the cameras and the head tracker is sent to the tracker and saved to the edf file. In certain experiments (for example the gaze-contingent windowing experiments that involve stimuli changing in real-time dependent on the subject's viewing behaviour), the tracker will also send information to the display computer during the trial. More detail on the system is available in the Eyelink II User Manual - it's worth reading.

3.2. What do I need to run an experiment?

Before you can run an experiment, you need three (equally important) things:

i. Your Python program!
ii. Stimuli - images / sounds / movies.
iii. A randomization file containing a unique stimulus presentation order for each subject of your experiment.

These files must be saved in a thoughtfully named directory on the display computer. You'll see that there is a directory called `experiments` on neocortex (the fancy new Mac that is our current display computer). You should create a directory in here for all files related to your experiment. The usual naming convention is to use your name or the experiment name, for example `alper_phasescram` contains everything Alper needs for his experiment using phase scrambled stimuli. The full filepath in this case is `/home/tracking/experiments/alper_phasescram/`. Within your directory, you should have your Python experiment script (normally named `experiment.py`), a directory (or more than one) containing your stimuli (for example `images/` should contain your images, while `audio/` should contain your sound stimuli), and your randomization file (a matlab .mat file, usually named `randorders.mat`). An edf file for each subject you track will automatically appear in your directory after each experiment you run, and it's good practice to make a directory called `edffiles` and to move them in there manually (this allows you to check that the file has been saved properly!). It's also good practice to back up your data in more than one location - you don't want to lose those edf files. Try to keep your experiment directory tidy so that it's clear which files you use for what purpose - somebody else might want to run your experiment some day, so be nice and make this as easy as possible.

We'll discuss the Python program in detail in the next chapter. Your stimuli will of course depend on your experiment, so we won't discuss them here. At present, typical image formats used are `png` or `bmp`; the typical audio format is `wav`.

The randomization file defines the order in which each of your subjects sees their stimuli. This file should contain a Matlab structure array of size number of subjects X number of trials. The choice of the fields you have in your structure array are extremely important. In the simplest case, you might just include a single field, `image`, which contains a number indicating which image your subject should see in each trial. So, if your structure array is called `rand`, and `rand(3,4).image` = 103, then in his/her 4th trial, your third subject will see image 103. When creating this structure array in Matlab, there are two things to keep in mind:

i. The general format of the structure array is:

   ```
   structarrayname(Subject_index, Trial_index).fieldname = value;
   ```

ii. The values should be either double or integers. If you don't have a clue what this means, don't worry - Matlab uses double by default.

If you also have auditory stimuli, you should have an `audio` field to dictate which auditory stimulus is presented for that subject in a given trial. It's standard practice to include a "dummy subject" as your first subject - this subject sees all stimuli in a non-randomised order, and is very useful for testing! It also sidesteps the discrepency in indexing conventions between Python (begins at 0) and Matlab (begins at 1).

Automated message-passing: One of the nicest advantages of using Pytrack is the automated message-passing that is built in. The fields of your randomisation mat file not only provide information about which stimuli should be presented at what time to whom, they are also automatically sent to the tracker and written to the edf file as meta-data about each trial. So, you can for example include an extra field called `condition` that explicitly records which condition that trial belonged to. This not only makes your experiment more transparent to other people in the lab who might need to understand it later when you're no longer here, it also makes life easier for you later on when you're analysing your data. We can't emphasise enough how nice a feature this is - use it!

3.3. Where's my data?

The eyetracking data and any meta-information collected during the experiment are written to an Eyetracker Data File (or edf for short). You supply the filename (which is limited to a maximum of 8 characters) before the experiment begins - see the next section for more details. This file remains on the tracker computer until the end of the experiment, at which point it should be automatically transferred to the display computer (more exactly, to the directory containing your experiment program). If something goes very wrong, the file remains on the tracker computer until it is overwritten. It's possible to boot the tracker computer in Windows and access the edf file, so don't lose hope if there are any problems during an experiment.

Chapter 4. Designing an Experiment

4.1. What do we need to do?

You should now have some idea of how the eyetracking system works during an experiment. You also have a good grasp of how to use Python. What we'll do now is to go through each step that is needed to run an experiment, so that you can have a clear idea of what is involved in a typical eyetracking program.

Several sample experiment scripts are available on neocortex - you'll find one (called `experiment.py`) in the folder `experiments/sample/`. These can easily be used as templates for writing your own experiments - just make a copy of the `sample/` directory and start from there.

4.2. Line by line experimental design

The first thing we'll do is to name the edf file for this subject. We'll do this using a pop-up dialog box, which prompts the experimenter to give a filename.

```
filename = Dialog("Please provide a filename, max 8 characters:")
```

Python has a dialog function, which is almost as straightforward as our pseudocode above - you must specify the request that will be displayed in the dialog box, and when the user inputs an answer (in this case the filename), it is assigned to our variable `filename`. You can use this method of acquiring input for other information that is needed for the experiment, for example:

```
sel = Dialog("Subject index:")
```

This time the experimenter should input a number, which is assigned to `sel`, the selected subject index. This will be used to read out the correct information from the randomization file. So let's specify which file contains the randomization.

```
rand = Matlab("randorders.mat")
```

Again, there is a Python library that deals with Matlab files (there are Python libraries to do almost anything you need). It is used exactly as written in our pseudocode, so now we have a two-dimensional array rand that is structured exactly as the Matlab structure array. We know which subject we're tracking, so we can extract the relevant row of trials for this run of the experiment:

```
trials = rand[sel]
```

The next important step is to initialise all of the components of the eyetracking system, namely the tracker and the display. The Pytrack library allows you to create instances of `Display` and `Tracker` objects, similarly to the instances of `Apple` we created earlier.

```
disp = Display()
track = Tracker(disp, filename)
```

Now we have a display object `disp`, which means we have control over the display computer's monitor. We also have a tracker object `track` which is associated with our display object and which will write to the specified edf file `filename`. The first stage of any eyetracking experiment is always the camera set-up and calibration. This is easily done:

```
track.setup()
```

Another important class method associated with the tracker object is used to pass meta-data to the edf file. This method is called `metadata` and takes two inputs, the meta-data key (the name for this piece of data) and its value. A prime example of meta-data is the index of the subject this data belongs to. This is sent to the tracker as follows:

```
track.metadata("SUBJECTINDEX", sel)
```

Now that everything is set up and ready to go, it's time to run the experiment, trial by trial. A `for` loop is the most obvious way of doing this:

```
for t in trials:
    # announce the trial to the tracker object
    track.trial(t)
    # tell the tracker to run a drift correction
    track.drift()
    # create an image-trial object
    # (this could be Trial.Audio or Trial.MPlayer)
    T = Trial.Image(t)
    T.run() # run this trial
```

That's really all there is to it. As hinted in the comment above, Pytrack has different flavours of trials - if your experiment involves showing images only to your subjects, then you need an image-trial. If you also present sounds, you need an audio-trial. This pseudo-code is of course a slightly simplified version of the real Python code - in a real script you pass more arguments to the trial constructor. `Trial.Image` and `Trial.Audio` take different inputs, and this is really the only point of difference between the types of trials that you need to worry about. We'll return to the issue of inputs to the trial a little later, but remember that you can always type `help(Trial.Image)` if you want to know more!

After all of the trials have been presented to the subject, it's time to finish the experiment. Again, this is a simple and painless process:

```
# shut down the display, releasing tracker's control of it
disp.finish()
# shut down the tracker, which includes saving your edf to the display computer
track.finish()
```

That's basically all that's involved in an experiment. A real script looks fundamentally the same as our pseudocode above, but there are some small differences. One important aspect is to provide more detailed information about which stimuli should be presented in each trial. As mentioned earlier, all of this information is contained in the randomization file. All that needs to be done then, is to translate the numerical value contained in a particular field of the rand structure array into a filename.