In this example we will use PsyParse to generate a dataframe and export to CSV. The dataframe will contain the following fields:

- image.on
- image.off
- image.filename
- probe.set
- probe.1.x
- probe.1.y
- probe.2.x
- probe.2.y
- user.response.value
- user.response.timestamp

In [1]:
```
# import PsyParse module
import psyparse

# create a logfile object using the included sample logfile
logfile = psyparse.Logfile(psyparse.sample_log_file())
```

A common first step after loading a logfile will be to check what variables exist in the logfile that are not already handled by **PsyParse**.

In [2]:
```
logfile.find_unmapped_variables()
```

```
The following unmapped variables were found:
image
text
text_2
```

Based on this output, it seems we will need to map three variables to **PsyParse** classes. To see what classes are available for mapping we can do the following:

In [3]:
```
psyparse.available_classes()
```

Out[3]:
```
['Image', 'Keypress', 'Probe', 'Trial']
```

It seems clear that variable 'image' will be mapped to class 'Image'. It is not clear without viewing the PsychoPy experiement, but in this case the two text variables represent probes that should be mapped to class 'Probe'.

In **PsyParse** variables are mapped in a variable map object and each individual mapping can be add to the variable map with a 3-tuple in the form of (variable name in logfile, variable name to be used in output, class to map variable).

In [4]:
```
# create an empty variable map object
variable_map = psyparse.VarMap()

# add mapping for image to name:image, class:Image
variable_map.add('image', 'image', 'Image')

# add mapping for text to name:probe.1, class:Probe
variable_map.add('text', 'probe.1', 'Probe')

# add mapping for text_2 to name:probe.2, class:Probe
variable_map.add('text_2', 'probe.2', 'Probe')

# set our logfile's variable map to our new map...
# after this we should have no more unmapped variables,
# we can verify this by checking that we find no unmapped
# variables
logfile.mapped_variables = variable_map
logfile.find_unmapped_variables()
```

```
No unmapped variables were found.
```

In addition to the variable mappings, a handler must be defined to handle the parsing. In this case, a frame handler will be used to construct a data frame that may be exported to a CSV file.

In [5]:
```
# create an empty handler
frame_handler = psyparse.handler.frame.Frame()
```

The frame handler requires a few parameters to be set. At its core, it is a directed tree structure, where nodes are selected around which to build records, and fields for said records are generated by traversing the tree in an upward direction and checking for nodes that match a given criteria.

First we must define which nodes to build our records around. In this case, I think it makes sense to build the records around user responses.

To define selected nodes we supply a list of 2-tuples where the 1st entry in the tuple represents the name of the attribute to query and the 2nd entry represents the value that our selected nodes should have for this attribute.

Lets first look at the attributes available for the Keypress class (user response) and then add ann appropriate entry to the selected nodes list.

```
In [6]: psyparse.attributes_for_class('Keypress')
```

```
Accessible attributes for Keypress are as follows:

        as_dict
        character
        klass
        log_text
        log_type
        name
        parent
        timestamp
        uuid
```

```
In [7]: # setting the nodes value as such means that we want
        # to build a record around every keypress
        frame_handler.nodes = [('name','keypress')] # the class name is returned in lowercase
```

Now that we have defined nodes to select, we need to define how the record should be build around these nodes. To do this we need to set the frame handler's variables with a list of 3-tuples where the first two entries of the 3-tuple define what node, neighboring the selected node, should be selected, and the last entry of the 3-tuple defines from which attribute we want to select a value. The form is as follows:

(attr of node to query, value of attr for matching node, attr for which to grab value for selected node)

I realize this sounds confusing, so let's step through an example of defining our fields in small chunks.

The first three fields in our desired data frame deal with the shown image:

- image.on
- image.off
- image.filename

Let's first see what attributes are available for the Image class and then define the connections for each of the three fields.

```
In [8]: psyparse.attributes_for_class('Image')
```

```
Accessible attributes for Image are as follows:

        as_dict
        filename
        klass
        log_text
        log_type
        name
        parent
        start
        stop
        timestamp
        uuid
```

```
In [9]: frame_handler.variables = [('name','image','start'),    # image.on
                                   ('name','image','stop'),     # image.off
                                   ('name','image','filename')] # image.filename
```

We also need to define a header that corresponds to the fieldnames in our frame so that the columns of our data frame are named appropriately.

```
In [10]: frame_handler.header = ['image.on', 'image.off', 'image.filename']
```

The next field that we want in our data frame is the **probe.set**. These data do not exist in the probes themselves. With the way this particular experiment was structured, this information can be found in the **rep** attribute of the **Trial** node that is the immediate parent of our selected node.

```
In [11]: psyparse.attributes_for_class('Trial')
```

```
Accessible attributes for Trial are as follows:

        as_dict
        index
        klass
        log_text
        log_type
        name
        parent
        rep
        start
        stop
        timestamp
        uuid
```

In [12]:
```python
# append this field to the variable list
frame_handler.variables += [('name','trial','rep')]

# append the field name to the header
frame_handler.header += ['probe.set']
```

The next fields in our data frame all relate to probe nodes.

- probe.1.x
- probe.1.y
- probe.2.x
- probe.2.y

As with the fields realted to images, let's examine the probe attributes and add the appropriate entries to the variable and header lists.

In [13]:
```python
psyparse.attributes_for_class('Probe')
```

```
Accessible attributes for Probe are as follows:

        as_dict
        klass
        log_text
        log_type
        name
        parent
        position
        start
        stop
        timestamp
        uuid
        x
        y
```

In [14]:
```python
# append fields to the variable list
frame_handler.variables += [('name','probe.1','x'), # probe.1.x
                            ('name','probe.1','y'), # probe.1.y
                            ('name','probe.2','x'), # probe.2.x
                            ('name','probe.2','y')] # probe.2.y

# append to the header list
frame_handler.header += ['probe.1.x','probe.1.y','probe.2.x','probe.2.y']
```

The last couple of fields in our desired data frame can be found in the user response nodes.

- user.response.value
- user.response.timestamp

Just as we have done with the others, let's examine the attributes of the keypress class, and add the appropriate entries to the variable and header lists.

In [15]:
```python
psyparse.attributes_for_class('Keypress')
```

```
Accessible attributes for Keypress are as follows:

        as_dict
        character
        klass
        log_text
        log_type
        name
        parent
        timestamp
        uuid
```

In [16]:
```python
# append fields to the variable list
frame_handler.variables += [('name','keypress','character'), # user.response.value
                            ('name','keypress','timestamp')] # user.response.timestamp

# append to the header list
frame_handler.header += ['user.response.value', 'user.response.timestamp']
```

Optionally, we can also ask our data frame to sort the records by one or more of our fields, as referenced by the corresponding entry in the header list. In this example case, I think it makes sense to sort records by **user.response.timestamp**.

In [17]:
```python
frame_handler.sort_by = ['user.response.timestamp']
```

Now we are finally ready to parse the logfile. We need to attach our new **frame handler** to our **logfile** instance and then we are ready to parse.

In [18]:
```
# attach the handler to the logfile
logfile.handler = frame_handler

# parse the logfile -- this may take a few seconds
logfile.parse()

# print the first few records of the data frame to make sure this worked!
logfile.handler.head()
```

```
image.on,image.off,image.filename,probe.set,probe.1.x,probe.1.y,probe.2.x,probe.2.y,user.response.value,user.r

14.7259,15.4759,003099.png,1,-210.0,202.0,-164.0,-231.0,y,16.7772
14.7259,15.4759,003099.png,2,-234.0,-1.0,134.0,-4.0,y,17.4776
14.7259,15.4759,003099.png,3,205.0,-142.0,-356.0,-218.0,y,17.9611
14.7259,15.4759,003099.png,4,-336.0,-304.0,-168.0,285.0,n,18.845
14.7259,15.4759,003099.png,5,296.0,-153.0,40.0,325.0,n,20.896
21.0288,22.0459,003049.png,1,-198.0,159.0,219.0,56.0,n,24.081
21.0288,22.0459,003049.png,2,95.0,-217.0,147.0,-348.0,y,25.4317
21.0288,22.0459,003049.png,3,-157.0,-118.0,-275.0,-327.0,n,27.4661
21.0288,22.0459,003049.png,4,-95.0,-67.0,-342.0,-146.0,n,28.2665
21.0288,22.0459,003049.png,5,217.0,242.0,-206.0,37.0,y,30.7511
```

Now all we have left to do is to export our data frame to a **CSV** file.

In [19]:
```
logfile.handler.write_to_csv('my-new-data-frame.csv')
```