Getting started with PyDatAnalysis

1. Make a new branch of repo.
2. Make a new ExpSpecific file in DataStandardize > ExpSpecific
   1. Probably easiest to duplicate last used and modify from there
   2. Main things it should contain are:
      1. Subclass of ExpConfigBase (For Experiment specific things)
      2. Subclass of SysConfigBase (For System specific things)
      3. Subclass of Exp2HDF (What is interacted with in program (i.e. nothing is ever asked of ExpConfig or SysConfig directly (I think) so overriding here should be sufficient to change behaviour)
3. Set new Exp2HDF as default in DatObject > Make\_Dat.py by changing default\_Exp2HDF to point to the new subclass of Exp2HDF
4. To load dats use an instance of DatHandler() which is a singleton which keeps track of what is already open.
   1. E.g.  
      ```  
      from DatObject.Make\_Dat import DatHandler  
      DH = DatHandler()  
      dat = DH.get\_dat(<datnum>, [optional args])   
      ```

What you can do with a dat:

A ‘dat’ as returned from DatHander.get\_dat(datnum) is the Python interface to a DatHDF in the directory specified in the Config classes. The DatHDF is where everything to do with the dat is saved. All information is looked for there first before being generated and saved. Note that for increased performance, most things are cached when read from the HDF file, so just changing the HDF directly will not necessarily update in a python runtime unless caches are cleared or the runtime is restarted.

The ‘dat’ itself has some normal attributes like ‘datnum’ and ‘date\_initialized’ but by itself is very minimalistic. Additional functions and information are generally stored in “DatAttributes” which are properties of the ‘dat’. (e.g. `dat.Data` gets you to the “Data” “DatAttribute”.)

In general, the DatAttributes are only generated when asked for, and if you ask for something which can’t be generated an error will be raised which hopefully will give some clue as to what was missing in order to create it (e.g. missing ‘i\_sense’ data).

Once created, they are automatically saved in the HDF file, so if you ask for the same thing again later, it will load from the HDF file rather than creating it again.

In general, it should always be safe to delete a whole DatAttribute (a ‘group’ in the HDF) from the HDF file (either through python or just in the HDF viewer) if you want to start that part again from scratch without losing everything else.

Currently the DatAttributes which you can use are:

* Data -- Should be able to load anything from experiment file, and also makes it easy to save new data. (Note: to avoid duplicating data for minor changes unnecessarily I use DataDescriptor's. Mostly you don't have to worry about it, but if for example you want the to change some raw data to be 10x larger so it's in units of nA for example, then it is more efficient to change only the data descriptor to say that when loading the data should be multiplied by 10 rather than saving a whole copy of the data with every value just 10x larger.
* Transition -- This makes it easier to do fitting and save multiple different fits. Use dat.Transition.get\_fit(...) to generate new fits or load old ones (if they don't exist they'll be created). Or dat.Transition.avg\_fit and dat.Transition.row\_fits will return fits using default fitting parameters. By default it will look for ‘i\_sense’ data (which is currently looked for in the experiment file under the names ‘cscurrent’ or ‘cscurrent\_2d’ based on the `ExpConfigBase.get\_default\_dat\_info()`… (I.e. if ‘cscurrent’ is saved in the Experiment file, ‘i\_sense’ will exist in the DatHDF).
* Entropy – Very similar to Transition but for entropy data.   
  Note: This will by default use the centers determined by SquareEntropy or dat.Transition.get\_centers() (which itself comes from dat.Transition.row\_fits) in order to average 2D entropy data into a 1D array for dat.Entropy.avg\_fit. This is done because the center positions from fits to transition data are usually more accurate than the center position from fits to entropy data.
* Logs – Quick and nicer access to all the things stored in sweeplogs of original experiment file.   
  Note: ‘dat.Logs.sweeplogs’ will return the full sweeplogs dict (after substitutions etc specified in config files). So if something is missing, you can look there to see if the thing is being stored under a different name than expected (i.e. someone may have changed how IGOR stores the data). In that case, either add a temporary fix to ConfigFiles (can be done for specific datnums) or add a new field following the pattern of all the rest (e.g. add to Logs.initialize\_minimum() and add a property right under Logs.\_\_init\_\_() and then re-run Logs.initialize\_minimum())
* Figures (this is pretty new and subject to change when I get working on the Dash app some more). The idea of this is to save figures in the DatHDF as well, that way figures can be passed between python runtimes easily, and also viewed from HDF more easily.
* AWG (Arbitrary Wave Generator of FastDAC) – This contains information about what the FastDAC AWG was doing during the scan, also has some helpful functions for making masks etc. For SquareWave AWGs specifically, see dat.SquareEntropy.square\_awg (which mostly copies info from AWG, but forces the AWG to be a 4 point wave (i.e. bunches all of the ramping parts into the main setpoints)).
* Square Entropy – This handles the conversion of charge sensor data into an entropy signal and stores all the relevant information along the way (i.e. setpoint averaged, cycled etc). It’s mostly comprised of three parts: Input, ProcessParams, Output. You can generate any of them by using SquareEntropy.get\_<thing>(…) and any params not specified will be generated automatically. Input is the data that is required to calculate square entropy stuff, process params are the things that affect how processing happens (i.e. where to start and finish averaging). Outputs contain all the different steps of processing and are saved to the HDF by default. The Inputs/ProcessParams are not saved to HDF by default because they are very quick to generate, but can be saved by specifying ‘save\_name’ and can be loaded by specifying ‘name’ (see docs for more info).   
  Note: No Entropy fitting is done here, that should be done from dat.Entropy (which will automatically use square entropy data if available). For more control, pass in the necessary arguments to dat.Entropy.get\_fit(…) (e.g. pass in the x, and entropy signal you want to use).