Purpose

Many scientific computing codes are designed to recieve some input data along with an input file that specify a (generally large) number of input options. In general, input files for these codes (or libraries) are somewhat cumbersome. Take Paramesh as an example: options are specified simply as a consecution of Fortran-style values to be read directly with little to no input sanitization or validation.

```
25 !  maxblocks
 2 !  ndim
 0 !  l2p5d
16 !  nxb
16 !  nyb
 1 !  nzb
...
```

In the case of some large-scale scientific computing codes, hundreds of options coupled with poor documentation and primitive IO can become a significant problem. The following was taken from a large-scale (intentionally anonymous) project:

```
F              #Record Error Norm
F              #Record Field Min Max
F              #Record Skin friction
1.            #Tinf
17.857142857142854        #pref
1.,0.,0.       #velVecInf
300.           #Twall
0.72           #prandtl
0.9            #turbulent prandtl number
1.4            #gamma
cp,62.5d0       #heat_cap  !1004.9157
T,450.           #Re_inf !!! -> runnning dimensional with Re_inf=1.d0
0.4            #CFL   0.193321595661993 (400/25) 1.38086854044280
0.6           #CFL_local
10             #Newton_iterations
F              #running with local CFL (not used)
0              #0: RK4, 1: BDF1, 2: BDF2
F              #steady flow
0.             #total time: T_total, if >0 then use total time instead if CFL specification
5              #lsf_levels_refine
0.1         #dxmin_refinement
0              #ini_refinement
11             #totlevels (maximum refinement allowed)
1              #bndy_refinement 1 0.01
1 0.01
0              #cont_file
T              #find irregular points
F              #domainTreatment
2             #input_LSF: -1: plane, 0: from file, 1: flat_plate, 2: circle/sphere
0.5
7.0 0.0 0.0
0              #nSpecialBCs
0            #nMotions
F              #moving domain
0              #noSampleSurface
```

To avoid excessive criticism, it is left to the reader to discern what possible problems arise from this type in input scheme.

It should be noted that many simulations performed by such codes are generally expensive, and small, un-checked mistakes can be costly. Consider, for example, any of the following situations:

- An input option has a minor typo in it. In the case of input options with named parameter, this could result in the parameter defaulting to a provided value without the user's knowledge.
- A number of input options should be the same or depend on some parameter, and a change to this parameter is neglected in some case.
- The reader fails and does not identify the mis-formatted parameter (improper exception handling)

In the author's experience, the list goes on.

PropTreeLib is designed to alleviate these common issues and provide a file format that is easy to organize, easy to read, and allows for a wide range of input types.

Features

Benefits of using a PropTreeLib (PTL) format include:

- Identification of misformatted parameters within tree-structure
- Simple and readable input format
- Parameterizeable input
- Support for many input types
- Automatic documentation of all input parameters
- Simple syntax for input bindings

Supported Input Types

PTL currently supports the following input types, although inheritace of the base `InputVariable` class allows for any input type:

- `PTLBoolean`: boolean value, valid inputs are: `true`, `True`, `t`, `T`, `false`, `False`, `f`, `F`
- `PTLDouble`: double-precision real number
- `PTLInteger`: integer
- `PTLString`: string
- `PTLEnum`: A string-enumeration that maps to an integer. Valud choices are passed as a formatted string, e.g. `"valuea:valueb:valuec"`
- `PTLDynamicDoubleArray`: Reads an array of an arbitrary number of doubles and allocates on heap. Maps to a double pointer and an integer for the number of elements.

All inputs are memory-safe and are disposed when the `PropertyTree` object goes out of scope.